ARISTOTLE UNIVERSITY OF THESSALONIKI

Department of Computer Science

Technical Report

"Populating Object-Oriented Rule Engines with the Extensional Knowledge of OWL DL Reasoners"

Georgios Meditskos and Nick Bassiliades

2008

Populating Object-Oriented Rule Engines with the Extensional Knowledge of OWL DL Reasoners

G. Meditskos[a], N. Bassiliades[a]

[a]*Department of Informatics, Aristotle University of Thessaloniki, 54124, Greece*

Abstract

The Web Ontology Language (OWL) has become the standard for defining and sharing ontologies on the Web, based on formal and machine processable semantics. In the present work, we define a framework for importing the (extensional) knowledge about OWL instances in Object-Oriented (OO) rule engines in order to develop practical, semantic web compliant, rule-based applications. We target at domains where ontologies are used as the means for exchanging knowledge among heterogeneous environments and serve as the back-end model of rule-based applications. To this end, we import the asserted and inferred OWL axioms of Description Logic (DL) reasoners in the KB of a native rule engine in order to be matched in the body of rules, acting as constraint model. The novelty of our approach is in the fact that, instead of the trivial mapping of DL reasoners' axioms into rule facts, we define a methodology that exploits the OO capabilities of OO rule engines. The idea is to take advantage of OO principles, such as class and attribute inheritance, and to define an OO model in such a way, so to preserve, in terms of OO relationships, the extensional semantics that have been inferred by the DL reasoner. In that way (a) the semantics of OWL ontologies are handled by sound and complete DL reasoners, and (b) instance constraints, such as instance class memberships or instance property values, are checked directly over the OO rule KB. We have implemented our methodology in the CLIPS-OWL library, an extension to the CLIPS production rule engine based on the Pellet DL reasoner that makes use of the COOL language in order to represent and check extensional OWL constraints.

Key words: object-oriented rule engines, OWL, description logic reasoners, production rules

1. Introduction

Nowadays, the Web has been evolved in a large repository of information and has become a useful means of communication and knowledge sharing. However, in order to exploit the Web to its full extent, information should become understandable not only to humans but also to machines. Towards this need, the Semantic Web initiative[1] works on standards, technologies and tools in order to give to the information a well-defined meaning, enabling computers and people to work in better cooperation. Ontologies can be considered as a primary key towards this goal

Email addresses: gmeditsk@csd.auth.gr (G. Meditskos), nbassili@csd.auth.gr (N. Bassiliades)

[1] http://www.w3.org/2001/sw/

since they provide a controlled vocabulary of concepts, each with an explicitly defined and machine processable semantics. The Web Ontology Language[2] (OWL) is the W3C recommendation for creating and sharing ontologies on the Web. It provides the means for ontology definition and specifies formal semantics on how to derive new information based on the Description Logic (DL) [1] knowledge representation formalism.

Rule-based systems have been extensively used in several applications and domains, such as e-commerce, personalization, games, businesses and academia. It is argued that the ability of manipulating and using semantically annotated information into rule systems is vital for both businesses and the successful proliferation of Semantic Web technologies, since it enables the already existing and well-known infrastructure of rule engines to gain access in the new evolution of Semantic Web.

In this work, we present our approach towards the utilization, rather than manipulation, of OWL ontological knowledge in rules engines, focusing on domains where ontologies are used to share knowledge between heterogeneous environments. Our framework is dedicated to OO rule engines and allows the population of their KB with the *extensional* knowledge (ABox) of OWL DL reasoners, which is the knowledge needed in rule-based applications that are defined *on top of the ontological knowledge*. Note the difference to the approaches that target at the *tight integration* of rules and ontologies that allow ontologies to be defined *on top of rules* and therefore, the schema-related knowledge is also important.

Our approach is based on the assumption that the DL reasoner's knowledge that is mapped on the rule engine should not be modified by rule programs, and therefore, there is no need for a constant and time-consuming interaction between the rule engine and the external DL reasoner for checking instance-related constraints. Although this hypothesis may seem a rather severe restriction, there are several applications and domains that follow this assumption, which we refer to as *isolated back-end ontological models*. Examples include rule programs that use ontology-based dictionaries, e.g. OWL/RDF WordNet[3], or ontologies that stem from database views, medical patient records or annotate resources, such as the Gene ontology[4]. In these domains, rules are used on top of the ontological vocabulary for building rule-based applications, for example, expert systems, rather than expressing richer ontological relationships that should be stored back in the initial ontology.

The mapping algorithms of the ontology ABox that we define in this paper are based on the utilization of OO principles for representing knowledge about OWL instances. Instead of the existing trivial approaches that map the asserted and inferred OWL axioms of DL reasoners on facts in the KB of the rule engine, such as the Jena [2] implementation of the DIG interface [3], we define a more sophisticated mapping methodology for OO rule engines, taking advantage of their OO capabilities. Our intention is to generate an OO model that preserves the extensional semantics that have been inferred by the DL reasoner, treating instance class memberships as object class type declarations. We are not affected by the semantic differences that exist between the OWL and the OO model[5] [4] (see also section 2.4), since *we do not use the OO model as an ontology inference mechanism*. Instead, the ontological reasoning is performed only by the DL reasoner and the OO model acts as an *indexing mechanism* of the ABox, resulting in a more compact and efficient representation of the ontology instance-related semantics in the rule KB

[2] http://www.w3.org/TR/owl-features/
[3] http://www.w3.org/TR/wordnet-rdf/
[4] http://www.geneontology.org
[5] http://www.w3.org/2001/sw/BestPractices/SE/ODSD/

than of using facts. In that way, we enable the OO rule engine to contain constraints about ontology instances in rule bodies that are checked by querying the OO KB, following the closed world (CWA) and unique name (UNA) assumptions of the database paradigm.

At this point we want to mention that the OO mapping algorithms we define in this paper (section 3) are generic and platform-independent, able to be implemented in any OO environment by utilizing the necessary language-dependent constructs, for example, Java interfaces for defining multiple inheritance in Java. However, in the present paper, we are focused on the practical application of the OO mapping of DL reasoners' KB on an OO rule engine and the advantages that stem from such an approach. To testify experimentally the soundness and effectiveness of our proposal, we have implemented our methodology in the CLIPS-OWL library, an extension to the CLIPS production rule engine that takes advantage of the OO language of CLIPS, namely COOL, and the inferencing capabilities of the Pellet DL reasoner [5]. In that way, we give the ability to CLIPS production rule programs to have access to a COOL OO model that is derived from an OWL ontology, using it to answer instance-related constraints.

The rest of the paper is structured as follows: in section 2 we present the background and the motivation relevant to our work. In section 3 we describe in detail the mapping algorithms of ontology concepts, roles and instances on OO classes, attributes and objects. In section 4 we elaborate on the architecture of the CLIPS-OWL library and we present the basic COOL syntax, exemplifying also on the way the COOL model can be used to represent extensional constraints. We present also experimental results that justify the effectiveness of the OO representation against the fact-based representation during constraint checking. Finally, in sections 5 and 6 we present related work and we conclude, respectively.

2. Background and Motivation

In this section, we present basic background related to the OWL and the DL reasoning paradigms, and we refer to the OO principles on which our transformation methodology is based. We present also the differences between OWL and OO semantics that hamper the direct use of the OO model as an OWL reasoning mechanism. Finally, we state our motivation in terms of our decision to follow the OO model for representing the extensional knowledge of DL reasoners, instead of the fact-based representation.

2.1. The Web Ontology Language

The Web Ontology Language (OWL) is the W3C recommendation for creating and sharing ontologies in the Web and its theoretical background is based on the Description Logic (DL) [1] knowledge representation formalism, a subset of predicate logic. It has emerged as the solution to the expressive limitations of RDF and RDF Schema[6] (RDFS) that offer the possibility to define only simple hierarchical relationships among concepts and properties, domain and range property restrictions and instances of concepts. OWL is a richer vocabulary description language for describing properties and classes, such as relations between classes (e.g., disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g., symmetry), and enumerated classes [6].

[6] http://www.w3.org/TR/rdf-mt/

```
<owl:Class rdf:ID="Human" />
<owl:Class rdf:ID="Man" >
   <rdfs:subClassOf rdf:resource="Human" />
</owl:Class>
<owl:Class rdf:ID="Boy" >
   <rdfs:subClassOf rdf:resource="Man" />
</owl:Class>
<Boy rdf:ID="peter" />
```

Figure 1: The XML/RDF syntax of an example ontology.

```
1: <Human> <rdf:type> <owl:Class> .
2: <Man> <rdf:type> <owl:Class> .
3: <Boy> <rdf:type> <owl:Class> .
4: <Man> <rdfs:subClassOf> <Human> .
5: <Boy> <rdfs:subClassOf> <Man> .
6: <peter> <rdf:type> <Boy> .
```

Figure 2: The N-Triples representation of Figure 1 (asserted triples).

An OWL ontology describes a domain of interest in terms of concepts, roles and instances and relationships among them. It is actually a finite set of DL axioms, such as axioms about concepts, concept inclusions ($A \sqsubseteq B$), role definitions, role inclusions ($R \sqsubseteq S$), concept assertions ($a : A$) and role assertions ($\langle a, b \rangle : R$), where A, B are concepts, R, S are roles and a, b are instances. These axioms can be divided into two categories, namely the TBox and the ABox of the ontology [6]. The TBox consists of the concept and role definitions/inclusions (terminological part), and the ABox of concept and role assertions (extensional part). Intuitively, the TBox refers to the schema of the ontology, whereas the ABox to the instances.

OWL comes in three flavors, namely OWL Full, OWL DL and OWL Lite that can be used according to the expressiveness that users desire during modeling. OWL Full does not impose any constraint to the OWL constructs that can be used and it is fully compatible, syntactically and semantically, with the RDF. However, the great degree of expressiveness does not guarantee computational decidability. OWL DL is a sublanguage of OWL Full and it is considered as the most expressive decidable sublanguage of OWL, restricting the application of some OWL and RDF constructs. OWL Lite further restricts the OWL DL sublanguage, and thus, has a restricted expressivity. However, it makes easier the development of related tools.

OWL is built upon RDF and RDFS and has the same syntax, the XML-based RDF syntax[7]. Figure 1 depicts the XML/RDF syntax of a simple ontology that defines three concepts, namely `Human`, `Man` and `Boy`, and an individual of the concept `Boy`, namely `peter`. A more machine processable syntax is the N-Triples format[8] that is a textual format for RDF graphs which stems directly from the RDF/XML syntax. More specifically, N-Triples is a line-oriented format where

[7] http://www.w3.org/TR/rdf-syntax-grammar/
[8] http://www.w3.org/TR/rdf-testcases/

each triple must be written on a separate line, and consists of a subject, a predicate, and an object, followed by a period. Figure 2 depicts the asserted triples in the N-Triple format of the ontology of Figure 1.

2.2. Description Logic Reasoning

OWL has its background on the DL knowledge representation formalism. For example, OWL DL and OWL Lite are based on $\mathcal{SHOIQ}(\mathcal{D})$ and $\mathcal{SHIN}(\mathcal{D})$, respectively. Thus, several OWL DL reasoners [5][7][8] implement DL algorithms, such as the tableaux-based algorithms [9]. These algorithms are based on trees whose nodes stand for elements of an interpretation domain and the inputs are assumed to be in the negation normal form, i.e. negation occurs in front of concept names only. A tableau algorithm employs a set of completion rules in order to infer new constraints or to generate new tree nodes and ensures termination using blocking algorithms, e.g. in the cases of existential quantifiers, such as $A \sqsubseteq \forall r.A$. Assuming that $\mathcal{KB} = (\mathcal{T}, \mathcal{A})$ is a DL knowledge base with the TBox $\mathcal{T}$ and the ABox $\mathcal{A}$, basic DL reasoning problems include:

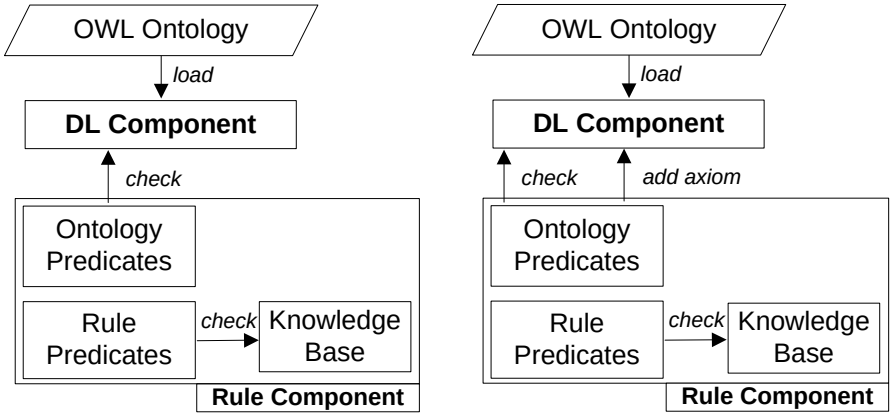
- **Concept equivalence.** Two concepts C and D are equivalent in $\mathcal{T}$ if and only if $\mathcal{T} \models C \sqsubseteq D$ and $\mathcal{T} \models D \sqsubseteq C$.
- **Concept subsumption.** A concept C is subsumed by D in $\mathcal{T}$ if and only if $C \sqcap \neg D$ is not satisfiable in $\mathcal{T}$.
- **Satisfiability.** A concept C is satisfiable in $\mathcal{T}$ if and only if C is not subsumed by $\bot$ (`owl:Nothing`) or $(\mathcal{T}, \{i : C\})$ is consistent.
- **Realization.** i is an instance of C according to the $\mathcal{KB}$ if and only if $(\mathcal{T}, \mathcal{A} \cup \{i : \neg C\})$ is not consistent.

2.3. Using OWL Ontological Knowledge with Rule Engines

In this section, we give a brief overview of the approaches for the utilization of OWL ontological knowledge with rule engines, which are classified according to the OWL reasoning mechanism that is employed.

2.3.1. Interfacing a DL Reasoner with a Rule Engine

One way of using OWL ontological knowledge in rule programs is to interface an external DL reasoner to the rule engine. These architectures, also known as *hybrid* [10], follow a modular architecture of two subsystems, each of which deals with a distinct portion of the knowledge base. More specifically, they combine the reasoning capabilities of the DL reasoning paradigm and the rule execution capabilities of a rule engine. The main characteristic of such architectures is that the rule and ontology predicates are strictly separated. The hybrid approaches are classified into *Bidirectional* and *Unidirectional*, according to whether the derived knowledge of rule programs flows from the rule module to the DL module or not. In Unidirectional frameworks (Figure 3a), the information flows only from the DL component to the rule component by allowing rule predicates to be used only in rule bodies as constraints, and thus, the DL reasoner's ontological knowledge remains unmodified throughout the execution of the rule program [11][12][13][14][15]. In Bidirectional frameworks (Figure 3b), DL predicates can be used both in the body and in the head of rules and thus, the ontological knowledge of the DL reasoner may be modified, allowing the development of ontologies on top of rules [16][17][18].



(a) Unidirectional framework (b) Bidirectional framework

Figure 3: Interfacing a DL component with a rule engine.

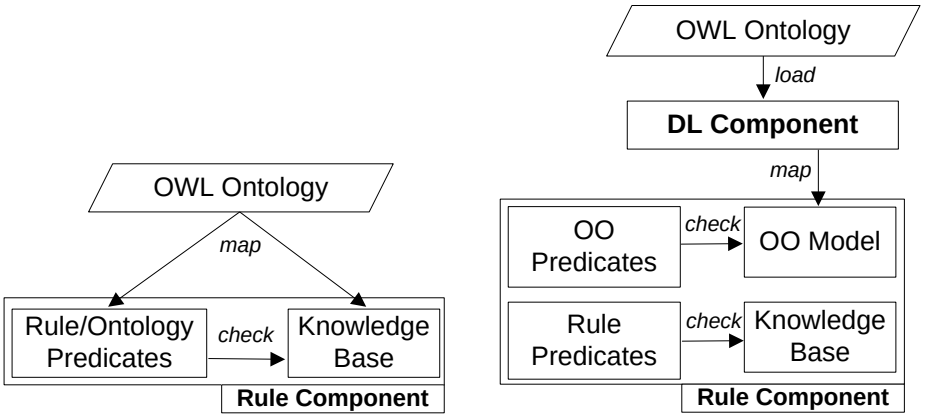
2.3.2. Mapping of OWL Semantics on Rules

In contrast to hybrid frameworks, there are approaches that do not employ a DL reasoner for OWL reasoning. Instead, they follow a methodology of mapping OWL ontologies on rules and facts in the rule engine (Figure 4a). These approaches, also known as *homogeneous* [10], treat rule and ontology predicates homogeneously, as a new single logic language. The general idea is that rules can use unary and binary predicates from the ontology (i.e., classes and properties), as well as predicates that occur only in rules (rules predicates). In order to maintain the decidability of the integrated language, there is usually a safety condition that restricts variables occurring in the head of a rule to those that occur in at least one positive rule predicate in the body of the rule. Intuitively, the OWL semantics are mapped into a rule formalism, e.g. Datalog rules, that coexist in the rule base (RB) with rule predicates, enhancing the expressivity. The homogeneous approaches can be used either for building rule programs on top of ontologies or ontologies on top of rules. Thus, a new reasoner is needed, able to handle the new homogeneous language that emerges [19][20][21].

2.3.3. Combination of Hybrid and Homogeneous Frameworks

Another way of using OWL ontological information in rule engines is to map the knowledge of the DL reasoner on the KB of the rule engine, such as the Jena implementation of the DIG interface that we have already mentioned, or the SWRLJessTab [22]. In such approaches, the OWL reasoning is performed by the DL reasoner and its knowledge is mapped on the knowledge representation formalism that the rule engine supports, which is usually triple-based facts. After the mapping of the ontological knowledge, there might be more interactions between the DL reasoner and the rule engine. If the ontology predicates are used only in rule bodies, then the ontological knowledge remains unchanged, preserving the initial semantics that have been inferred by the DL reasoner. Therefore, there is no need for an interaction between the two components. Otherwise, if the ontology predicates are used in rule heads, the ontological knowledge is modified and the DL reasoner should be employed again in order to reason over the modified ontology and to map again the ontology.

Our approach differs from the above architecture in the fact that we follow a different knowl-



(a) Homogeneous (b) OO Unidirectional Homogeneous

Figure 4: Homogeneous and OO Unidirectional Homogeneous frameworks.

edge representation paradigm to represent OWL axioms, and we refer to it as an *OO Unidirectional Homogeneous* framework (Figure 4b). More specifically, we use a DL reasoner for OWL reasoning (unidirectional hybrid-like) and we map the reasoner's KB into the KB of the OO rule engine without using further the DL component for answering queries. Instead, the queries are answered directly by the rule KB (homogeneous-like) that are expressed in the form of OO ontology predicates in the body of the rules as constraints. Therefore, in our framework, *the rule engine is not used for ontology reasoning* but it is used only for executing the rule program. As we explain in section 2, we follow such an approach since we are interested in the efficient execution of rule programs in isolated ontological environments following the OWA and UNA, and in the preservation of the initial semantics of the mapped DL reasoner's extensional knowledge.

2.4. Object-Oriented Programming

The Object-Oriented Programming (OOP) is a programming paradigm that allows the modeling of domains of interest in terms of classes with attributes and objects. It does not separate the data from the operations, as the traditional procedural languages impose (e.g. the C language) but it groups operations and data into modular units called objects that programmers may combine into structured networks to form a complete program. The OO modeling of information lies closer to the way programmers model a real-world domain by categorizing objects and concepts of the world into classes, attaching to them appropriate attributes. Basic OOP techniques include *inheritance*, *encapsulation*, *message passing*, *abstraction* and others [23]. The intuitiveness of the OO modeling paradigm has been embedded also in rule engines (OO rule engines), such as Drools[9], CLOS [24] and CLIPS [25], allowing the representation of knowledge in terms of OO modeling principles and the use of rules on top of these OO models.

At first sight, there are strong commonalities between the primitives that are used in OWL and the OOP paradigm. The concepts, roles and instances of OWL can be considered as the

[9]http://www.jboss.org/drools/

corresponding equivalents to the classes, attributes and objects in OOP. However, there are fundamental semantic differences between the two modeling paradigms that we briefly analyze in the following [26][4].

- **Concepts and classes.** In OWL, a concept is considered as a set of instances, in contrast to the OOP where a class defines a class type. The set-oriented nature of OWL accounts for a greater degree of expressiveness during concept definition, enabling the utilization of semantics related to sets, such as concept *equivalence* (`owl:equivalentClass`), *union* (`owl:unionOf`), *intersection* (`owl:intersectionOf`), *disjointness* (`owl:disjointWith`) or *complement* (`owl:complementOf`).

- **Roles and attributes.** In OWL, a role is a first-class citizen which is defined as a standalone entity without a specific class, whereas the attributes in OOP are defined locally to the class where they belong. Since OWL roles are actually instances, they have roles too. In that way, a role can be defined as the *inverse* (`owl:inverseOf`) of another role, or it might be defined *equivalent* (`owl:equivalentProperty`) to some other role, notions that cannot be directly represented in OOP.

- **Instances and objects.** In OWL, an instance might belong to more than one concepts, whereas in OOP an object can have only a single direct class type. Furthermore, OOP does not support the notion of identical objects of OWL (`owl:sameAs`) and therefore it follows the UNA.

To sum up, there are major semantic differences between OWL and OOP that makes incomplete the direct mapping from the one modeling paradigm to the other, especially from OWL to the OO model, since OWL has richer semantics. For example, class equivalence cannot be represented in the OO model, since in the latter, subclass graph cycles are forbidden (see concept equivalence is section 2.2). Furthermore, OWL follows the OWA, able to handle incomplete information, in contrast to OOP where unknown information means false information. Therefore, *the OO model can not be used by itself as a reasoning mechanism on OWL ontologies*.

However, in the present paper, we show how basic OO principles, such as class and attribute inheritance, can be used as an efficient indexing mechanism of the instance-related OWL ontology semantics that are computed by a DL reasoner. The inadequacy of preserving the terminological semantics of OWL ontologies in the OO model does not affect the importance of our methodology, since, in rule-based application, we are usually interested in the extensional knowledge of a domain, which we are able to fully represent in our OO model. Note that our approach is not a framework for manipulating ontologies with rules, but we rather use the ontological information as the base model in order to build rule-based applications.

2.5. Motivation

In this section we state our motivation in terms of our decision to define a framework dedicated to isolated ontological environments and to use the OO model for representing the extensional knowledge of OWL DL reasoners.

2.5.1. Suitability in Isolated Ontological Environments

Both the hybrid and the homogeneous frameworks we have described in section 2.3 have fundamental weaknesses, regarding their application in isolated ontological environments, that we summarize in the following.

Hybrid Approaches. The hybrid frameworks need to make external calls to DL reasoners in order to answer the ABox constraints that are present in the body of rules in unidirectional frameworks, or in order to classify and realize the DL KB due to changes in the ontological knowledge in bidirectional approaches. Therefore, in order to implement a hybrid framework, there is the need to modify the language, or even the architecture of the rule engine, in order to embed the results of the external calls in rule activation and execution. Furthermore, such a modification affects the performance in terms of rule execution time, since a communication overhead is introduced between the rule engine and the DL reasoner.

We argue that this nature of hybrid approaches does not fit in isolated ontological environments. Since the ontological knowledge is not altered by the rule program, there is no need for a constant communication between the rule engine and the DL reasoner. For that reason, we follow the approach of mapping the KB of the DL reasoner on the rule engine. In that way (a) any constraint is answered directly by the rule engine's KB without introducing an extra communication overhead with an external component, and (b) there is no need to modify the rule engine since the functionality of the two modules is not interleaved, using the DL reasoner only in the beginning of the operation in order to create the OO model. Bear in mind that, by mapping the DL reasoner's KB on the rule engine, any instance constraint is checked following the UNA and CWA, in contrast to the DL reasoning paradigm that follows the open world assumption (OWA) and the UNA may not be employed. However, the UNA and CWA are not always undesirable features and their suitability depends on the application domain (see section 2.5.2), such as in the database paradigm.

Homogeneous approaches. The great advantage of the homogeneous frameworks is that they allow richer ontology relationships to be modeled that cannot be expressed directly in OWL, enhancing the expressivity. However, the ability of the homogeneous systems to manipulate ontologies via rules is not needed in isolated ontological environments. Actually, the coexistence of the ontology inference rules with the rule program in the same RB might affect rule execution performance, since the rule engine has to manage a larger RB, or it might perform undesirable ontological modifications. The latter is a major drawback, especially in cases where the rule program operates over critical ontologies where modifications are unacceptable, such as a rule-based application whose knowledge derives from an ontology about medical records of patients where the rule-based application should be defined on top of the knowledge without modifying it. In our approach, the fact that the RB contains only the rule program, together with the OO nature of the mapped knowledge, makes harder the modification of the mapped knowledge. Moreover, the use of a DL reasoner guarantees OWL reasoning completeness, in contrast to the rule-based OWL reasoning paradigm that is followed by the homogeneous systems, which is based on the partial mapping of OWL ontologies on inferencing rules [27][28].

2.5.2. Unique Name and Closed World Assumptions

In hybrid frameworks, any instance constraint is checked at runtime by the DL reasoner, following the OWA and supporting the notion of identical instances (`owl:sameAs`). In our approach, we use a DL reasoner only in the beginning of the operation to reason at once over the ontology and to transform its KB into the OO model that the rule engine uses to answer instance-related queries. Therefore, the constraints about ontology instances are "reduced" to queries about objects and their values in the OO model. Thus, we follow the UNA, considering that all objects are different from each other, and the CWA, matching only known, to the OO KB, objects. Therefore, our framework is suitable in environments where the UNA and CWA

are requirements, i.e. in domains where we are interested more in identifying KB inconsistencies rather than supporting further inferencing, such as in database applications [27]. In that way, we can use the rule engine in order to define *integrity constraints*, that is, constraints about the ABox. For example, the ontology that contains the axioms $\texttt{Person} \sqsubseteq \exists \texttt{hasSSN}.\texttt{SSN}$ and $\texttt{george} : \texttt{Person}$ is satisfiable in OWL, even if we do not define an SSN number for george (open-world semantics). Such cases can be easily treated as inconsistencies by defining custom rules in the rule engine. On the other hand, the DL reasoning paradigm is oriented more to the nature of Web, where usually the notion of incomplete information is present and the OWA is a requirement.

2.5.3. Object-Oriented Modeling

A common way of representing the asserted and inferred OWL ontology axioms in a rule engine is to store the ontology triples in the form of facts. Note that we do not refer to the mapping of ontologies on rule engines that can be performed with rules (homogeneous approaches). We refer to the representation of the ontological knowledge in the KB of a rule engine, after the application of the reasoning process. In that way, the knowledge of the ontology of Figure 1 can be mapped on the triples/facts of Figure 2 and 5 (the triples of Figure 5 are derived via reasoning).

The limitation of the triple-based representation is that it is not able to exploit any form of semantics that could potentially exist in the environment where the mapping will take place. In that way, all the triples should be explicitly stated, following a brute force approach with increased space requirements. We argue that the semantics of an OO environment can be effectively used in order to represent the instance-related DL reasoner axioms in an OO model that embeds the notion of class subsumption transitivity.

More specifically, in an OO environment, if A is subclass of B and B is subclass of M, then A is also subclass of M, without the need to explicitly state it. Moreover, if a is an object of class A, then it is by default an object of classes B and M. Therefore, there is no need to explicitly state the knowledge related to instance class membership, as well as knowledge related to role inheritance. In that way, both the asserted and inferred triples of our example in Figure 1 can be mapped on the OO model that is depicted in Figure 6, where the IS-A denotes OO subclass relationship and the INSTANCEOF is used for object class type declaration. The advantage becomes more obvious if we consider large scale ontologies with many instances and subclass relationships. For example, in the fact-based representation, all the instance class membership relationships (rdf:type) should

```
7: <Boy> <rdfs:subClassOf> <Human> .
8: <peter> <rdf:type> <Man> .
9: <peter> <rdf:type> <Human> .
```

Figure 5: The N-Triples format of the inferred triples in Figure 1.

```
Man IS-A Human
Boy IS-A Man
peter INSTANCEOF Boy
```

Figure 6: The OO representation of the semantics of the example ontology.

be stated, without exploiting the class subsumption. Note that we are only interested in ABox relationships. Therefore, following the semantics of the OO model of Figure 6, `peter` is an object of the classes `Boy`, `Man` and `Human`, as the OWL semantics impose via the triples 6, 8 and 9 of Figure 2 and Figure 5.

Furthermore, in the OO model, the related information to a specific object is integrated into object's attribute values, as well as into class information, according to the position of the object's class type in the class hierarchy. Therefore, all the related information about the same object is integrated as one information unit and it can be retrieved directly by sending messages to the object of interest. In the fact-based representation, the knowledge is scattered across the memory as facts and the query procedure needs at runtime to traverse many triples. Finally, we are able not only to represent the knowledge in an OO manner, but to form also queries for checking constraints following the widely used and well-known techniques of the OOP, instead of plain triples.

3. Object-Oriented Mapping Methodology

In this section, we describe in detail the OO mapping procedure of the KB of DL reasoners on the OO model. We present firstly the syntax we use to represent OO relationships and the basic OO principles on which our methodology is based.

3.1. Object-Oriented Syntax and Principles

We use the notation $\langle m \text{ INSTANCEOF } A\rangle$ to denote the initialization of the object m with the class type A, $Obj(A)$ to denote the set of objects of class type A, $Att(A)$ to denote the set of attributes that are accessible by the objects of class A, and $\langle A \text{ IS-A } B\rangle$ to denote that A is a direct or indirect subclass of B.

Proposition 1. An object can be initialized only once, using a single class type declaration, i.e. the definitions $\langle m \text{ INSTANCEOF } A\rangle$ and $\langle m \text{ INSTANCEOF } B\rangle$ cannot simultaneously hold, for $A \neq B$. In other words, there cannot be two objects with the same ID in an OO KB.

Proposition 2. The OO subclass relationship holds transitivity. For example, if A is subclass of B and B is subclass of M, then A is also subclass of M, that is, $\langle A \text{ IS-A } B\rangle \wedge \langle B \text{ IS-A } M\rangle \rightarrow \langle A \text{ IS-A } M\rangle$.

Proposition 3. An object belongs to the set of objects of its class type. For example, if $\langle m \text{ INSTANCEOF } A\rangle$, then $m \in Obj(A)$. The reverse relationship does not necessarily hold due to Proposition 4.

Proposition 4. An object belongs also to the sets of objects of all the superclasses of its class type. For example, if $\langle m \text{ INSTANCEOF } A\rangle$, then $\forall B|\langle A \text{ IS-A } B\rangle : m \in Obj(B)$.

Proposition 5. Subclass graph cycles are forbidden. For example, the $\langle A \text{ IS-A } B\rangle$ and $\langle B \text{ IS-A } A\rangle$ hierarchical relationships cannot simultaneously hold.

Proposition 6. A class may have more than one direct superclass (multiple direct inheritance).

Proposition 7. A class attribute may have multiple declarations in different classes, since it is not a first class citizen, as in OWL.

Proposition 8. An object has access to the attributes of its class type. For example, if $\langle m$ INSTANCEOF $A\rangle$, then $m.P$ denotes the set of values of the object m for the attribute P, $\forall P \in Att(A)$.

Proposition 9. The class attributes are inherited to subclasses and they can be accessed by the subclasses' objects. For example, if $P \in Att(B)$ and $\langle A$ IS-A $B\rangle$, then the definition $m.P$ is valid for every object of A.

3.2. Mapping Algorithms

In this section, we describe the mapping algorithms of ontology concepts, roles and instances of a DL reasoner's KB on the OO model. Bear in mind that the OO model is unable to preserve the complete terminological relationships (TBox) of OWL ontologies, such as concept equivalence (Proposition 5). The goal of the mapping procedure is to preserve the extensional only knowledge of the ontology, that is, instance class membership relationships and instance role values, in terms of object class type declarations and object attribute values, respectively. To achieve this goal, the mapping procedure consists of two phases: (a) the generation of an OO schema of classes with attributes that is not necessarily compliant with the TBox ontology semantics, but it is specially defined in order to preserve the ABox semantics, and (b) the definition of the objects and their attribute values with respect to the previously defined OO schema.

For the legibility of the presentation of the mapping algorithms, we refer to the ontology example in Table 1 that describes the university domain. More specifically:

$a1$. A chair is also a professor
$a2$. A person who is the head of a department is also a chair
$a3$. A person is a man or a woman
$a4$. All persons are individuals and vice versa
$a5$. A department is also an organization
$a6$. All organizations are institutes and vice versa
$a7$. The domain of the role `isHeadOf` is the concept `Individual`
$a8$. The domain of the role `isHeadOf` is the concept `Professor`
$a9$. The range of the role `isHeadOf` is the concept `Organization`
$a10$. The instance `csd` belongs to the `Department` concept
$a11$. The instance `nick` belongs to the `Person` concept
$a12$. `nick` is the `headOf` the computer science department

In the rest of the discussion we assume that $\mathcal{C}, \mathcal{R}$ and $\mathcal{I}$ are the sets of the asserted and inferred named concepts, roles and instances, respectively, of the DL reasoner's KB after the reasoning procedure over an ontology, and $\mathcal{C}_o$ and $\mathcal{I}_o$ are the sets of classes and objects, respectively, of the OO model. Note that there is not a corresponding set $\mathcal{R}_o$ in the OO model, since attributes are not first class citizens (Proposition 7). The o subscript is used throughout the paper to denote OO-related constructs.

3.2.1. Mapping the Ontology Subsumption Hierarchy

An OWL ontology may contain concept subsumption relationships of the form $A \sqsubseteq B$, or concept definitions based on set operator semantics, such as *intersection*, *union*, *equivalence* and *complement*. The TBox inferencing procedure of a DL reasoner is responsible for computing the ontology subsumption hierarchy, that is, the complete subclass relationships among the ontology

#	OWL Axioms (DL Syntax)
$a1$	$\texttt{Chair} \sqsubseteq \texttt{Professor}$
$a2$	$\texttt{Chair} \equiv \texttt{Person} \sqcap \exists \texttt{isHeadOf}.\texttt{Department}$
$a3$	$\texttt{Person} \equiv \texttt{Man} \sqcup \texttt{Woman}$
$a4$	$\texttt{Person} \equiv \underline{\texttt{Individual}}$
$a5$	$\texttt{Department} \sqsubseteq \texttt{Organization}$
$a6$	$\texttt{Organization} \equiv \underline{\texttt{Institute}}$
$a7$	$\top \sqsubseteq \forall \texttt{isHeadOf}^{-}.\underline{\texttt{Individual}}$
$a8$	$\top \sqsubseteq \forall \texttt{isHeadOf}^{-}.\texttt{Professor}$
$a9$	$\top \sqsubseteq \forall \texttt{isHeadOf}.\texttt{Organization}$
$a10$	$\texttt{csd} : \texttt{Department}$
$a11$	$\texttt{nick} : \texttt{Person}$
$a12$	$\langle \texttt{nick, csd} \rangle : \texttt{isHeadOf}$

Table 1: Example ontology axioms.

concepts. Our mapping procedure is based on the subsumption hierarchy that is computed by the DL reasoner and defines a class in the OO model for each named concept in C, along with specially defined OO subclass relationships. Consequently, the anonymous and the restriction classes (`owl:Restriction`) do not participate in the mapping procedure, since their semantics, which are relevant to concept classification, instance realization and consistency checking, are handled by the DL reasoner. Bear in mind that the OO model is not able to perform queries referring to non-existing classes or objects in the KB (CWA). For that reason, there is no need to map any anonymous ontology concept on the OO model.

Definition 1. Each named ontology concept $A \in C$ of the DL reasoner is mapped on a corresponding OO class $A_o \in C_o$, which is denoted as $A \curvearrowright A_o$.

In that way, each concept of Table 1 will be mapped on an OO class, except for the anonymous restriction $\exists \texttt{isHeadOf}.\texttt{Department}$. This existential restriction will be used by the DL reasoner in order to perform concept classification and instance realization based on the values of the role `isHeadOf`. We should mention here that, although for each named concept A in C there is the corresponding A_o class in C_o, the reverse relationship does not necessarily hold. As we explain later, there are cases where a class A_o in C_o does not have a corresponding concept A in C, that is, $\exists A_o \in C_o$, such that $\nexists A \in C$ with $A \curvearrowright A_o$. Therefore, in general, we have that the size of the concept and class sets is not the same, that is, $|C_o| \neq |C|$. This fact does not affect our approach, since we are interested in the ontology ABox semantics and the OO schema is used as the means for defining later the objects.

Furthermore, OWL semantics impose that every concept is subsumed by the built-in `owl:Thing` class ($\top$), that is, $\forall A \in C, A \sqsubseteq \top$. Therefore, we define $\texttt{owl:Thing}_o$ as the superclass of all the OO classes (OO hierarchy root), even if the `owl:Thing` concept is not refrerenced by an axiom of the mapped ontology.

Preprocessing of equivalent concepts. OWL semantics impose that two concepts A and B are equivalent, if and only if a mutual subsumption relationship holds between them, that is, $A \equiv B \Leftrightarrow A \sqsubseteq B \wedge B \sqsubseteq A$. In that way, all the equivalent classes have the same class extension

(the same set of instances), which is denoted as $CEXT(A) = CEXT(B)$. However, such mutual subclass relationships cannot be modeled in an OO environment (Proposition 5). In order to overcome this limitation, we perform a preprocessing of the named equivalent concept sets that derive from DL reasoning before applying the mapping procedure, assigning *delegators*.

Algorithm 1. *For each set E of named equivalent concepts that the DL reasoner infers, we arbitrary choose a concept $A \in E$ as the delegator concept, such that $\forall M \in E, dlg(M) = A$. For each concept $N \in C$ that does not have equivalent concepts, we define that its delegator is the concept itself, that is, $dlg(N) = N$.*

To exemplify, consider the $a4$ and $a6$ axioms of Table 1. The DL inferencing procedure results in the named equivalent concept sets $E_1 = \{\texttt{Person}, \texttt{Individual}\}$ and $E_2 = \{\texttt{Organization}, \texttt{Institute}\}$. With the Algorithm 1 we assign a delegator to each set, let the `Individual` for the former and the `Institute` for the later (depicted underlined in Table 1), such that $dlg(\texttt{Person}) = dlg(\texttt{Individual}) = \texttt{Individual}$ and $dlg(\texttt{Organization}) = dlg(\texttt{Institute}) = \texttt{Institute}$. Moreover, $dlg(\texttt{Chair}) = \texttt{Chair}$, since `Chair` does to have a named equivalent concept. The purpose of this preprocessing, which strongly affects the rest of the mapping procedure, will be made clear during the instance mapping algorithms in section 3.2.3.

Ontology subclass relationships. The subsumption hierarchy is determined after DL reasoning in the form of direct subclass relationships among the concepts of the DL reasoner's KB. In the rest of the discussion, we refer to such direct subclass relationship between two concepts A and B in the KB of the DL reasoner as $A \ll B$, denoting that A has been inferred to be a direct subclass of B.

Algorithm 2. *For each subclass relationship $M \ll N$ that the DL reasoner infers, we define that $\langle A_o$ IS-A $B_o\rangle$, where $A = dlg(M)$, $B = dlg(N)$, $A \curvearrowright A_o$ and $B \curvearrowright B_o$.*

The Algorithm 2 maps ontology concepts on OO classes, *allowing only delegator concepts to participate in OO subclass relationships*. More specifically, there are four mapping cases considering the $M \ll N$ subclass relationship:

1. *if $dlg(M) = M$ and $dlg(N) = N$ then $\langle M_o$ IS-A $N_o\rangle$*. This is the case when none of the two ontology concepts have equivalent classes, or they are both the delegators of some equivalent concept sets. In this case, the OO subclass relationship should contain the two corresponding OO classes M_o and N_o to the ontology concepts ($M \curvearrowright M_o$ and $N \curvearrowright N_o$).
2. *if $dlg(M) = A$ and $dlg(N) = N$ then $\langle A_o$ IS-A $N_o\rangle$*. This is the case where the concept M belongs to an equivalent concept set for which the concept A is the delegator. Therefore, the concept M is substituted with the corresponding OO class to its delegator concept A ($A \curvearrowright A_o$) in the OO hierarchical relationship.
3. *if $dlg(M) = M$ and $dlg(N) = B$ then $\langle M_o$ IS-A $B_o\rangle$*. This is an analogous case to 2.
4. *if $dlg(M) = A$ and $dlg(N) = B$ then $\langle A_o$ IS-A $B_o\rangle$*. In this case, both the M and N concepts belong to equivalent concept sets for which the concepts A and B are the delegators. Therefore, they are substituted with the corresponding OO classes A_o and B_o to their delegators in the OO hierarchical relationship ($A \curvearrowright A_o$ and $B \curvearrowright B_o$).

To exemplify, the TBox inferencing procedure on the ontology of Table 1 computes, among others, that $\texttt{Chair} \ll \texttt{Professor}$ (from axiom $a1$) and that $\texttt{Department} \ll \texttt{Organization}$

and Department $\ll$ Institute (from axioms $a5$ and $a6$). The first relationship belongs to case 1 above, since dlg(Chair) = Chair and dlg(Professor) = Professor and therefore, $\langle \text{Chair}_o$ IS-A $\text{Professor}_o \rangle$, where Chair $\curvearrowright$ Chair_o and Professor $\curvearrowright$ Professor_o (in the rest of the discussion, for simplicity, we omit any $A \curvearrowright A_o$ notation, assuming that A_o is always the OO class of the corresponding ontology concept A). The second relationship falls into case 3, since dlg(Department) = Department and dlg(Organization) = Institute and therefore, $\langle \text{Department}_o$ IS-A $\text{Institute}_o \rangle$, although the direct subclass relationship defines explicitly that Department $\ll$ Organization. Finally, the third relationship belongs to case 1, since dlg(Department) = Department and dlg(Institute) = Institute, and therefore $\langle \text{Department}_o$ IS-A $\text{Institute}_o \rangle$, similarly to the previous case. Therefore, the $\langle \text{Department}_o$ IS-A $\text{Organization}_o \rangle$ relationship is not implemented in our example.

Concept intersection. The semantics of concept intersection impose multiple subsumption relationships among the involving concepts, that is, if $A \equiv A_1 \sqcap A_2 \sqcap \ldots A_n$, then $A \sqsubseteq A_i$, $1 \leq i \leq n$. The DL reasoning procedure results in multiple direct subclass relationships, that is, $A \ll A_i, 1 \leq i \leq n$, that we map following the Algorithm 2. In that way, the $a2$ axiom of Table 1 is mapped on the $\langle \text{Chair}_o$ IS-A $\text{Individual}_o \rangle$ relationship, since Chair $\ll$ Person, dlg(Chair) = Chair and dlg(Person) = Individual.

Concept union. Concept union imposes multiple subsumption relationships among the involving concepts, that is, if $A \equiv A_1 \sqcup A_2 \sqcup \ldots \sqcup A_n$, then $A_i \sqsubseteq A, 1 \leq i \leq n$. The DL reasoning procedure results in multiple direct subclass relationships, that is, $A_i \ll A, 1 \leq i \leq n$, that we map following the Algorithm 2. In that way, the $a3$ axiom is mapped on the $\langle \text{Man}_o$ IS-A $\text{Individual}_o \rangle$ and $\langle \text{Woman}_o$ IS-A $\text{Individual}_o \rangle$ relationships, since Man $\ll$ Person, Woman $\ll$ Person, dlg(Man) = Man, dlg(Woman) = Woman and dlg(Person) = Individual.

Complement concepts. The semantics of concepts that are defined as the complement of others are used only by the DL reasoner in order to determine ontology inconsistencies, without affecting the mapping procedure (in a similar way to concept restrictions). The same holds for other relevant OWL constructs, such as owl:disjointWith, which denotes concepts whose instances cannot belong simultaneously in the class extensions of the disjoint concepts, and therefore, these concepts must not have any hierarchical relationship.

Concept equivalence. Since we are not allowed to define mutual subclass relationships among classes in the OO model, we follow a special mapping algorithm based on delegator concepts.

Algorithm 3. *Let the set E of named equivalent concepts, where A is the delegator concept, that is,* $\forall N \in E, dlg(N) = A$*, and let* D_o *be the set that contains the corresponding OO classes to the concepts in E, that is,* $D_o = \{M_o | M \curvearrowright M_o, \forall M \in E\}$*. We define then that* $\langle A_o$ IS-A $M_o \rangle$*,* $\forall M_o \in D_o - \{A_o\}$*, where* $A \curvearrowright A_o$*.*

The goal of Algorithm 3 is to define the OO class of a delegator concept to be the subclass of all the OO classes of the delegator's equivalent concepts. To exemplify, the $a4$ axiom is mapped on the $\langle \text{Individual}_o$ IS-A $\text{Person}_o \rangle$ OO hierarchical relationship, since we have defined that the Individual concept is the delegator of the equivalent concept set {Person, Individual}. Furthermore, the $a6$ axiom is mapped on the $\langle \text{Institute}_o$ IS-A $\text{Organization}_o \rangle$ OO hierarchical relationship, since we have defined that the Institute concept is the delegator of the equivalent concept set {Organization, Institute} (see Algorithm 1). In section 3.2.3 we explain

the rationale behind this algorithm and how it is used in order to represent the ontology concept equivalence semantics in the OO model.

3.2.2. Mapping Ontology Roles

The object (`owl:ObjectProperty`) and datatype (`owl:DatatypeProperty`) ontology roles are mapped on class attributes based on their domain and range restrictions. The domain restrictions are used in order to determine the class where the attribute must be defined and the range restrictions are used to restrict the type of the values that the attribute can accept. The mapping procedure assumes that all the attributes are defined in such a way, so to be able to take more than one values in the form of a list.

Definition 2. For each ontology role $P \in \mathcal{R}$, there is at least one attribute definition P_o in an OO class $A_o \in C_o$, denoted as $P \Rrightarrow A_o[P_o \twoheadrightarrow Type]$, where $Type$ is the set of the type constraints of P_o. If P is an object property, then the $Type$ of P_o is a set of classes of the form $\{T_1, T_2, \ldots, T_n\}$ that denotes that an object value, let o_v, is allowable to be stored in P_o, if it satisfies the class type constraints $o_v \in Obj(T_1) \vee o_v \in Obj(T_2) \vee \ldots \vee o_v \in Obj(T_n)$. If P is a datatype property, then the $Type$ of P_o is the set of type constraints of the form $\{T_1, T_2, \ldots, T_n\}$ that denotes that a data value, let d_v, is allowable to be stored in P_o if it satisfies the types constraint $type(d_v) = T_1 \vee type(d_v) = T_2 \vee \ldots \vee type(d_v) = T_n$.

Therefore, the `isheadOf` property of Table 1 has at least one $\texttt{isHeadOf}_o$ attribute definition in an OO class. The OO class will be determined based on the role's domain restrictions (axioms $a7$ and $a8$) and the type will be determined based on the range restrictions (axiom $a9$). In the following, we analyze the algorithms for determining the class where an attribute should be defined in the OO model, as well as the attribute type constraints.

Domain restriction. OWL semantics impose that if a role has not an explicit domain restriction, then it is accessible by all instances and therefore it is assumed that its domain restriction is the `owl:Thing` concept. Therefore, the set of the domain restrictions of a role is never empty. Furthermore, a role is allowed to have more than one concept as domain restrictions that act as class intersection.

Algorithm 4. *Let a role $P \in \mathcal{R}$ with a single domain restriction A. If A belongs to a named equivalent concept set E, then we define P_o in every class in E (allowable by Proposition 7), that is, $P \Rrightarrow B_o[P_o \twoheadrightarrow Type], \forall B_o \in E_o$, where E_o is the set that contains the corresponding OO classes to the concepts in E, that is, $E_o = \{B_o | B \curvearrowright B_o, \forall B \in E\}$. Otherwise, P_o is defined directly in A_o, that is, $P \Rrightarrow A_o[P_o \twoheadrightarrow Type]$.*

To exemplify on the semantics of Algorithm 4, assume that the axiom $a8$ was not defined and the `isHeadOf` role had only the `Individual` concept as domain restriction. Recall also that, due to the axiom $a4$ and Algorithm 3, we have that $\langle \texttt{Individual}_o$ IS-A $\texttt{Person}_o \rangle$. If we define the $\texttt{isHeadOf}_o$ attribute only in the $\texttt{Individual}_o$ class, as it is imposed by the $a7$ axiom, then it will not be inherited to the $\texttt{Person}_o$ class, because $\texttt{Person}_o$ is superclass of $\texttt{Individual}_o$ (Proposition 9). Since $\texttt{Person} \equiv \texttt{Individual}$, the two concepts have the same roles, and thus, the objects of the corresponding OO classes should have access to the same attributes. Therefore, we define the $\texttt{isHeadOf}_o$ multislot in every OO class of E_o, that is, in both $\texttt{Person}_o$ and $\texttt{Individual}_o$ classes.

In the case where a role has more than one domain restrictions, OWL semantics impose that the domain of the role is finally the intersection of the domain concepts.

Algorithm 5. *Let a role $P \in \mathcal{R}$ with a domain restriction concept set D, with $|D| \geq 2$. We create the set D' with the most specific concepts of D ($D' \subseteq D$), that is, $\forall A \in D', \nexists B \in D'$ such that $B \sqsubseteq A$. If $|D'| = 1$, we follow the Algorithm 4. If $|D'| \geq 2$, we create a class T_o, such that $\langle T_o$ IS-A $dlg(N_o)\rangle, \forall N_o \in D'_o$, where D'_o contains the corresponding OO classes to the concepts in D', that is, $D'_o = \{N_o | N \curvearrowright N_o, \forall N \in D'\}$. P_o is finally defined in T_o, that is, $P \Rrightarrow T_o[P_o \twoheadrightarrow Type]$.*

In the case of more than one domain concepts, we preprocess the set D in order to obtain the set D' that contains only the most specific concepts, that is, the concepts in D that do not subsume other concepts in D. If $|D'| = 1$, then we follow the Algorithm 4 that handles single domain restrictions. If $|D'| \geq 2$, a new OO class T_o is created as a subclass of all the corresponding OO classes to the delegators of the concepts in D', and P_o is defined finally in T_o. In that way, only objects of the T_o class have access to the P_o attribute, as the semantics of OWL impose. This is the case we mentioned in Definition 1 that we have $|C| \neq |C_o|$, since T_o is an auxiliary class and it is present only in the OO model, without having a corresponding ontological concept in C. However, as we have already mentioned, we are interested only in preserving the ABox semantics of the ontology.

To exemplify on Algorithm 5, consider the domain restriction axioms $a7$ and $a8$ for the role `isHeadOf`. We have that $D = \{$`Individual`, `Professor`$\} = D'$, since there is not a subsumption relationship between the two domain concepts. Therefore, we create an auxiliary class Aux_o and we define it as the subclass of the corresponding OO classes to the delegators of the domain concepts `Individual` and `Professor`, that is, $\langle$`Aux`$_o$ IS-A `Individual`$_o\rangle$ and $\langle$`Aux`$_o$ IS-A `Professor`$_o\rangle$, since dlg(`Individual`) = `Individual` and dlg(`Professor`) = `Professor`. In that way, the attribute is finally defined in the `Aux`$_o$ class, that is, `isHeadOf` $\Rrightarrow$ `Aux`$_o$[`isHeadOf`$_o$ $\twoheadrightarrow Type$].

Range Restriction. The range restrictions are used in order to restrict the value types of attributes. We should mention here that any ontology inconsistency is determined and handled by the DL reasoner. Therefore, the reasoner is responsible for reporting any improper value type in an role or to perform further inferencing in order to handle role value inconsistencies. For that reason, the restriction of the type of attributes in the OO model is done only for completeness and documentation issues, since, due to the fact the ontology remains unchanged, there will be no further inconsistencies captured by the OO model attribute type constraints.

Let D be the set of the range restrictions of a role P. If P does not have a range restriction, then:

- if P is an object property, then we define $D = \{$`owl:Thing`$\}$, allowing any object to be considered as a value.
- if P is a datatype property, then we define $D = \varnothing$, allowing data values of any type.

Algorithm 6. *Let an object property $P \in \mathcal{R}$ with a single range restriction B. If B belongs to a named equivalent concept set E, then we define $P \Rrightarrow A_o[P_o \twoheadrightarrow Type]$, where A_o is the attribute definition class that is determined by Algorithms 4 and 5, and the $Type$ set contains all the corresponding OO classes to the ontology concepts in E, that is, $Type = \{N_o | N \curvearrowright N_o, \forall N \in E\}$. If B does not have equivalent concepts, then we define directly that $P \Rrightarrow A_o[P_o \twoheadrightarrow \{B_o\}]$, where $B \curvearrowright B_o$.*

More specifically, if the range restriction concept B of an object property has named equivalent concepts, then we should restrict the attribute to take objects of any of the corresponding

OO classes to the equivalent concepts, and of course, of their subclasses. For that reason, the $Type$ set should contain all the corresponding OO classes to the equivalent concepts of B (see Definition 2). Otherwise, if the range concept B does not have equivalent concepts, the attribute is restricted to take objects of the B_o class only (and its subclasses).

For example, the `isHeadOf` role (axiom $a9$) is mapped as $\texttt{isHeadOf} \Rrightarrow \texttt{Aux}_o[\texttt{isHeadOf}_o \twoheadrightarrow \{\texttt{Institute}_o, \texttt{Organization}_o\}]$, since it has more than one domain restrictions (represented by the $\texttt{Aux}_o$ class) and $\texttt{Organization} \equiv \texttt{Institute}$. Therefore, it can take as values only objects of the classes $\texttt{Organization}_o$ or $\texttt{Institute}_o$.

Algorithm 7. *Let an object property $P \in \mathcal{R}$ with a range restriction set D, with $|D| \geq 2$. We create the set D' with the most specific concepts of D ($D' \subseteq D$), that is, $\forall A \in D', \nexists B \in D'$ such that $B \sqsubseteq A$. If $|D'| = 1$, we follow the Algorithm 6. If $|D'| \geq 2$, we create a class T_o, such that $\langle T_o$ IS-A $dlg(N_o)\rangle, \forall N_o \in D'_o$, where D'_o contains the corresponding OO classes to the concepts in D', that is, $D'_o = \{N_o | N \curvearrowright N_o, \forall N \in D'\}$. In that way, P_o is restricted to take objects of class T_o, that is, $P \Rrightarrow A_o[P_o \twoheadrightarrow \{T_o\}]$ (A_o is determined by the Algorithms 4 and 5).*

More specifically, in the case of more than one range restrictions, we follow a similar approach to Algorithm 5 in order to determine the set D'_o with the most specific range concepts. Then, we generate a system class T_o subclass of all the classes in D'_o and the $Type$ set is finally defined to contain the T_o class. In that way, the attribute is allowed to take as values objects that belong to the T_o class.

Algorithm 8. *Let a datatype property $P \in \mathcal{R}$ with a range restriction set D. If $D = \varnothing$, then we define that $P \Rrightarrow A_o[P_o \twoheadrightarrow \{\}]$. If $D = \{T\}$, we define $P \Rrightarrow A_o[P_o \twoheadrightarrow \{datatypeMap(T)\}]$. If $|D| \geq 2$, we define $P \Rrightarrow A_o[P_o \twoheadrightarrow \{datatypeMap(T), \forall T \in D\}]$.*

More specifically, in the case where a datatype property has no type restrictions, we do not restrict the type, allowing the attribute to take any data value. If the ontology role has a single type constraint, we define the set $Type$ to contain the corresponding OO type ($datatypeMap$) to the ontology datatype restriction, according to the available type restrictions of the OO environment where the mapping takes place. For example, the `xsd:int` datatype can be mapped on the `INTEGER` type restriction of CLIPS. In the case where more than one datatype restrictions exist in the ontology, then we map all of them on OO type constraints and we allow P_o to take values of these datatypes only.

3.2.3. Mapping Ontology Instances

In this section, we describe the mapping algorithms of ontology instances into the OO model, based on the OO schema of classes and attributes that is generated by the previous steps. Note that the ontology TBox semantics are not preserved by the concept and role mapping algorithms. For example, only the $\texttt{Individual}_o$ class is subclass of the $\texttt{Person}_o$ class and not vice versa, as it should be due to the concept equivalence semantics. With the mapping algorithms of concepts and roles the goal is to generate the necessary infrastructure on which the objects will be defined, allowing instance-related information to be queried during the development of rule-based applications.

Definition 3. Each ontology instance $m \in \mathcal{I}$ of the DL reasoner is mapped on a corresponding OO object $m_o \in \mathcal{I}_o$, which is denoted as $m \Rsh m_o$.

In that way, the `nick` and `csd` ontology instances (axioms $a10$ and $a11$) would have the corresponding $\texttt{nick}_o$ and $\texttt{csd}_o$ objects in the OO model. The instance mapping procedure has two basic characteristics:

- The set of objects of an OO class is the same to the corresponding class extension of an ontology concept, following the UNA. Each instance class membership relationship of the ontology is mapped on an appropriate object class type declaration in the OO model. This mapping is defined in such a way, so that each ontology instance has a corresponding OO object that can be retrieved by querying the corresponding OO classes to the ontology concepts where the instance belongs in the ontology, that is, $\forall m \in CEXT(A), \exists m_o \in Obj(A_o)$, where $m \Rsh m_o$ and $A \curvearrowright A_o$.
- The instance $m \in \mathcal{I}$ and its corresponding object $m_o \in \mathcal{I}_o$ have the same values in the corresponding roles/attributes, following the UNA. Each role value of an ontology instance is mapped on an appropriate value on the corresponding attribute to the ontology role of the corresponding object to the ontology instance, that is, $\forall \langle m, m' \rangle : P, \exists m_o$, such that $m'_o \in m_o.P_o$ (object properties) or $\forall \langle m, v \rangle : P, \exists m_o$, such that $v \in m_o.P_o$ (datatype properties), where $m \Rsh m_o, m' \Rsh m'_o$ and $P \Rrightarrow A_o[P_o \twoheadrightarrow Type]$.

Note that the above relationships are true only if the UNA is taken into consideration, since the instance equality of OWL cannot be represented in the OO model. In that way, even if the DL reasoner infers that two ontology instances m and i are equal ($\langle i, m \rangle$: `owl:sameAs`), these instances will be mapped as two distinct objects on the OO model in order to be able to query both of them. These two objects, however, would be different only in their object ID, having the same class type declaration, the same attributes and the same values in their corresponding attributes.

Object class type declarations. The class type declarations of objects in the OO model *refer only to delegator classes*. This is an important characteristic of the instance mapping procedure, since it enables to represent the concept equivalence semantics that are related to the instance class memberships, imposing that if $A \equiv B$, then $\forall m \in CEXT(A), m \in CEXT(B)$. Therefore, in terms of OO principles, if $A \equiv B$ then $\forall m_o \in Obj(A_o), m_o \in Obj(B_o)$, where $A \curvearrowright A_o, B \curvearrowright B_o$ and $m \Rsh m_o$. We are able to model such relationships based on the semantics of OO class hierarchies (Proposition 4).

Algorithm 9. *Let the set D denote the concepts where the ontology instance $m \in \mathcal{I}$ belongs to (instance class membership relationships), after the DL realization procedure. If $D = \{K\}$, then we define m_o as an object of the corresponding OO class to K's delegator, that is, $\langle m_o$ instanceOf $A_o \rangle$, where $dlg(K) \curvearrowright A_o$. Otherwise ($|D| \geq 2$), we create the set D' with the most specific concepts of D ($D' \subseteq D$), that is, $\forall A \in D', \nexists B \in D'$ such that $B \sqsubseteq A$. If $|D'| = 1$, we follow the previous case ($D = \{K\}$), since the set D' contains only a single concept. Otherwise ($|D'| \geq 2$), we create or reuse a class T_o, such that $\langle T_o$ is-a $dlg(N_o) \rangle, \forall N_o \in D'_o$, where the set D'_o contains the corresponding OO classes to the concepts in D', that is, $D'_o = \{N_o | N \curvearrowright N_o, \forall N \in D'\}$, and we define $\langle m_o$ instanceOf $T_o \rangle$.*

More specifically, in the case where the instance m belongs to a single ontology concept K, the class type of the object m_o is the corresponding OO class to K's delegator. The semantics of this mapping are the following: suppose that the concept K belongs to a named concept

equivalence set E, whose delegator is the concept W. Therefore, the W_o class ($W \curvearrowright W_o$) would be defined as subclass of all the corresponding OO classes of its equivalent concepts (Algorithm 3), that is, $\langle W_o$ IS-A $A_o\rangle, \forall A_o \in E_o - \{W_o\}$, where $E_o = \{M_o | M \curvearrowright M_o, \forall M \in E\}$. In that way, any object of W_o is also an object of all its superclasses, that is, of all its equivalent classes (Proposition 4). Therefore, by forcing the objects to belong always to delegator classes, we achieve that all the OO classes of an equivalent concept set have the same objects, preserving the concept equivalent semantics in terms of class type declarations in the OO model. If K has no equivalent concepts, then the semantics are still preserved, since $dlg(K) = K$.

In the case where an instance belongs to multiple concepts, we preprocess the set D of concepts in a similar way to Algorithms 5 and 7. In this case, however, we firstly check if an appropriate T_o class already exists due to Algorithms 5 and 7, and reuse it, defining that $\langle m_o$ INSTANCEOF $T_o\rangle$. Otherwise, we create one from scratch.

To exemplify, the DL realization procedure classifies the `nick` instance in the `Chair`, `Person`, `Individual` and `Professor` ontology concepts (axioms $a2$, $a8$ and $a12$), and therefore, D = {`Person`, `Individual`, `Chair`, `Professor`} in Algorithm 9. In order to preserve these instance class membership semantics of the DL procedure during the instance mapping procedure, $\texttt{nick}_o$ should be an object of classes $\texttt{Person}_o$, $\texttt{Individual}_o$, $\texttt{Chair}_o$ and $\texttt{Professor}_o$ in the OO model, which is achieved as follows: since the subsumption hierarchy of DL reasoning imposes that `Chair` $\sqsubseteq$ `Person`, `Chair` $\sqsubseteq$ `Individual` and `Chair` $\sqsubseteq$ `Professor`, we have that `Chair` is the most specific concept in D, that is, D' = {`Chair`}. Therefore, we define that $\langle \texttt{nick}_o$ INSTANCEOF $\texttt{Chair}_o\rangle$, since $dlg(\texttt{Chair}) = \texttt{Chair}$ (Algorithm 9). In that way, $\texttt{nick}_o \in Obj(\texttt{Chair}_o)$, $\texttt{nick}_o \in Obj(\texttt{Person}_o)$, $\texttt{nick}_o \in Obj(\texttt{Individual}_o)$ and $\texttt{nick}_o \in Obj(\texttt{Professor}_o)$, since $\langle \texttt{Chair}_o$ IS-A $\texttt{Individual}_o\rangle$, $\langle \texttt{Individual}_o$ IS-A $\texttt{Person}_o\rangle$ and $\langle \texttt{Chair}_o$ IS-A $\texttt{Professor}_o\rangle$, as we have described in sections 3.2.1 and 3.2.2, preserving the instance class membership relationships (the set D) of DL reasoning. It is worth mentioning that the equivalence semantics between the `Person` and the `Individual` concepts are always preserved, since their corresponding OO objects would always be defined in the $\texttt{Individual}_o$ class, which is the delegator concept.

Moreover, the `csd` instance is classified in the `Department`, `Institute` and `Organization` concepts, and therefore, D = {`Department`, `Institute`, `Organization`} ($a10$ and $a12$ axioms). Since `Department` $\sqsubseteq$ `Organization` and `Department` $\sqsubseteq$ `Institute`, then D' = {`Department`} and we define that $\langle \texttt{csd}_o$ INSTANCEOF $\texttt{Department}_o\rangle$, since $dlg(\texttt{Department}) = \texttt{Department}$. In that way, $\texttt{csd}_o \in Obj(\texttt{Department}_o)$, $\texttt{csd}_o \in Obj(\texttt{Institute})$ and $\texttt{csd}_o \in Obj(\texttt{Organization})$, since $\langle \texttt{Department}_o$ IS-A $\texttt{Institute}_o\rangle$ and $\langle \texttt{Institute}_o$ IS-A $\texttt{Organization}_o\rangle$, as we have described in sections 3.2.1 and 3.2.2, preserving the instance class membership relationships that are computed by DL reasoning.

Instance role values. If the value of a role is an instance (`owl:ObjectProperty`), then the corresponding object of the OO model is inserted in the corresponding attribute. Otherwise (`owl:DatatypeProperty`), the data value is directly inserted. In the former case, we exploit the DL reasoning procedure about same instances in order to insert also the same instances (objects) of an instance value, that is, if $\texttt{owl:sameAs}(m, n)$ and $\langle y, m\rangle : P$, then we map also the $\langle y, n\rangle : P$ axiom on the OO model, following the UNA.

Algorithm 10. *Each $\langle m, y\rangle : P$ axiom is mapped by inserting the value y in the attribute P_o of the m_o object, where $m \Rsh m_o$ and $P \Rrightarrow A_o[P_o \twoheadrightarrow Type]$. If y is an instance, then $y_o \in m_o.P_o$, where $y \Rsh y_o$. If y is a data value, then $y \in m_o.P_o$.*

Note that the semantics that are relevant to special OWL roles, such as *transitive*, *symmetric*, *inverse roles*, etc., are handled by the DL reasoner. Therefore, if $\langle m, y \rangle : P$ and P is a symmetric role, then the DL reasoner would also infer that $\langle y, m \rangle : P$ and the mapping algorithm will map both relationships as values in the corresponding OO objects y_o and m_o.

4. The CLIPS-OWL Library

The CLIPS-OWL library is a prototype implementation of our OO mapping algorithms, using the Pellet DL reasoner and mapping its KB on the COOL OO language of the CLIPS production rule engine. In this section, we present the basic architecture of CLIPS-OWL, the COOL syntax for defining classes and objects, as well as the way objects can be matched in the body of CLIPS production rules. For legibility purposes, we present also the COOL syntax of the example ontology of Figure 1 and a simple example of using CLIPS production rules with ontology-related OO constraints. Finally, we present experimental results that justify the efficiency of the OO representation against the typical fact-based representation.

4.1. Architecture

The architecture of CLIPS-OWL is depicted in Figure 7. The core of the library is the *Object-Oriented Model Generator (OOMG)*, the module that generates the OO model in the COOL language of CLIPS based on the inferencing capabilities of the Pellet DL reasoner. OOMG consists of three submodules, namely the *Class Generator*, the *Attribute Generator* and the *Object Generator*, in accordance to the three classes of algorithms we described in sections 3.2.1, 3.2.2 and 3.2.3 that are relevant to the mapping of concepts, roles and instances, respectively.

A typical usage scenario of the CLIPS-OWL library involves the following steps:

1. The loading of the OWL ontology in the Pellet DL reasoner in order to perform TBox and ABox reasoning. During this phase, Pellet performs concept classification and instance realization, as well as it checks the ontology for inconsistencies (total materialization procedure).
2. If the first step is successful, that is, no ontology inconsistencies have been detected, then OOMG starts the mapping procedure using the Pellet Java API. The Class and Attribute Generators cooperate, since the attributes are defined inside class definitions (Proposition 7). The Object Generator acts independently and it is applied after the application of the other two modules.
3. The output of the CLIPS-OWL library is the corresponding COOL model of the ontological knowledge that has been loaded in Pellet. This model can then been loaded in the CLIPS production rule engine, together with the rule program that uses the generated OO model to answer object constraints.

From the architecture of the CLIPS-OWL library in Figure 7, it is clear that the DL and the rule components are not tightly connected. The functionality of the two components is not interleaved, as it happens in native hybrid environments where the rule engine needs to call the DL component at run-time. In that way, we are able to use a DL reasoner and the rule engine without any modification of their architecture.

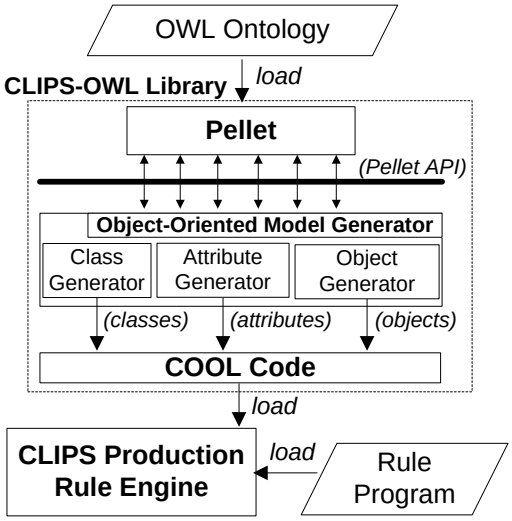


Figure 7: The architecture of CLIPS-OWL.

4.2. The CLIPS Engine and the COOL Syntax

CLIPS is a RETE-based production rule engine written in C that was developed in 1985 by NASA's Johnson Space Center and it has undergone continual refinement and improvement ever since. Today it is widely used throughout the government, industry and academia. Some of its main features are stability, speed, extensibility and low cost (public domain software).

One of the most interesting capabilities of CLIPS is that integrates the production rule paradigm with the OO model, which can be defined using the COOL (CLIPS Object-Oriented Language) language of CLIPS. In that way, classes, attributes and objects can be matched on the production rule conditions (LHS), as well as to be altered on rules actions (RHS), presenting a fully dynamic behavior

The semantics of CLIPS are the usual production rule semantics: rules whose condition is successfully matched against the current data are triggered and placed in the conflict set. The conflict resolution mechanism selects a single rule for firing its action, which may alter the data. Rule condition matching is performed incrementally, through the RETE algorithm.

Concerning the OO model, the semantics of CLIPS are the usual of an OOP environment, supporting also the principles of section 3.1. CLIPS supports *abstraction*, *inheritance*, *encapsulation*, *polymorphism* and *dynamic binding*. Some of the main OO features of CLIPS include:

- **Single and multiple inheritance.** It is possible to define multiple direct super-classes for a CLIPS class. In that way, each object of a subclass belongs also to all the superclasses.
- **Attribute inheritance.** Class attributes are inherited to the subclasses as well.
- **Message-based functionality.** Each object is able to receive and send messages to other objects. Every action related to objects is performed via messages, such as the insertion of values into object attributes (`put` message) or for reading the attribute values of an object (`get` message).

```
(defclass <name>
      (is-a <superclass-name>+)
      (<multislot>* <constraint>*))

(make-instance [<name>] of <class>
      [(<multislot> <values>)]*)
```

(a) Defining classes, multislots and objects

```
(object <attribute-constraint>*))

<attribute-constraint> ::=
  (is-a <constraint>) |
  (name <constraint>) |
  (<multislot-name> <constraint>*)
```

(b) Matching objects in rule conditions

Figure 8: Basic syntax of the COOL language of CLIPS.

- **Single and multifield attributes.** Each attribute can be defined as a single slot, taking only a single value, or as a multifield (multislot) attribute, able to take more than one values in a form of a list.

Figure 8a depicts the basic COOL syntax for defining classes and objects. A class is defined by specifying the name, one or more superclasses, and zero, one or more multislot (attribute) definitions with type constraints. We will be interested in the `type` and `allowed-classes` constraints only. The former is used to restrict the type of datatype attributes, whereas the latter denotes the class type of the objects that a multislot can take, using the `INSTANCE-NAME` type constraint. As we will see later, the built-in `USER` class of COOL must be the root class of the class hierarchy. An object is defined by specifying the name inside brackets ([and]), which is actually the object ID, the class type and any multislot value. Figure 8b depicts the object pattern syntax that is used in CLIPS production rules in order to match objects in the body of rules. An object is matched if satisfies the `is-a` class constraint, the name constraint and the constraints about multislot values. A variable, which is denoted as `?x` (a multislot variable is denoted as `$?x`), can be used at any place, except for the multislot name[10].

4.2.1. An Example COOL Model

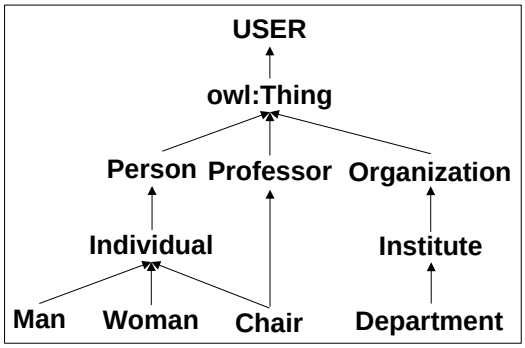
To exemplify on the COOL syntax and semantics, we present in Figure 9a the COOL model that is generated by using the CLIPS-OWL library to map the example ontology axioms of Table 1, based on the Pellet inferencing capabilities. For legibility purposes, we present in Figure 9b the hierarchical relationships among the classes of the COOL model.

The OO class hierarchy of Figure 9b encapsulates the following semantics, which are compliant with the extensional knowledge that is inferred by the Pellet reasoner, always in terms of OO relationships. Note that the o subscripts have been omitted.

[10]http://clipsrules.sourceforge.net/

```
(defclass Chair (is-a Professor Individual))
(defclass Professor (is-a owl:Thing))
(defclass Person (is-a owl:Thing)
   (multislot isHeadOf (type INSTANCE-NAME)
      (allowed-values Organization)))
(defclass Individual (is-a Person))
(defclass Man (is-a Individual))
(defclass Woman (is-a Individual))
(defclass Institute (is-a Organization))
(defclass Department(is-a Institute))
(defclass Organization (is-a owl:Thing))
(make-instance [computer_science] of Institute)
(make-instance [nick] of Chair
   (isHeadOf [computer_science]))
```

(a) The COOL model of the example ontology in Figure 1



(b) The hierarchical relationships

Figure 9: The example COOL code.

1. Each object of class `Man` is also an object of classes `Individual` and `Person`, as axioms $a3$ and $a4$ impose in Table 1.
2. Each object of class `Woman` is also an object of classes `Individual` and `Person`, as axioms $a3$ and $a4$ impose.
3. Each object of class `Chair` is also an object of classes `Professor`, `Individual` and `Person`, as axioms $a1$, $a2$ and $a4$ impose.
4. Each object of class `Individual` is also an object of class `Person`, as OWL axiom $a4$ imposes. Note that each object of class `Person` is not an object of class `Individual`. We have explained in section 3.2.3 how this inconsistency is treated during object definitions by forcing objects to belong always to *delegator* classes. In that way, every object of class `Person` is always defined in class `Individual` (delegator class).
5. Each object of class `Department` is also an object of classes `Institute` and `Organization`, as OWL axioms $a5$ and $a6$ impose.
6. Each `Institute` object is also an object of class `Organization`, as OWL axiom $a6$ imposes (same semantics to case 4).

7. Every object belongs to class `owl:Thing`.

4.3. Representing Instance Constraints in Object Patterns

Table 2 depicts the object patterns that can be used to represent common ontology constraints about instances in CLIPS production rule bodies.

4.3.1. Instance class membership

The instance class membership restrictions are reduced in OO queries about the class type of objects. Such constraints can be used either to retrieve the set of objects that satisfy a class constraint, or to check if a particular object satisfies a class constraint. More specifically, class intersection is treated by defining multiple object patterns in the body of the rule that must be satisfied by an object $?x$ (Table 2 (a)). Class union is treated by allowing an object to satisfy at least one (`OR` operator |) class constraint (Table 2 (b)). In both cases, instead of a variable object name, a concrete object name can be used in order to check if the specific object satisfies the class constraint. Note that, due to the delegator-oriented OO mapping procedure, a class constraint involves also the objects of its equivalent classes.

It is worth mentioning that we do not use the conjunctive object pattern syntax `(is-a C1& ... &Cn)` for intersection, as we do in union with the corresponding disjunctive syntax, for the following reason: the only case where an object can belong to more than one classes is when the classes have a hierarchical relationship, since an object in the OO model can have only a single class type declaration. In that way, the class constraint `(is-a C1& ... &Cn)` is valid only if there is a class, subclass of all the `Cn` classes, otherwise an error occurs during the processing of the rule by CLIPS. In order to avoid such errors, we follow the approach depicted in Table 2 (a), using multiple object patterns.

Moreover, negation-as-failure (NAF) can be used in order to match an object that does not satisfy the class constraints over the current OO KB (Table 2 (c) and (d)). Note that, due to the

#	DL Constraint	COOL Object Pattern

(*a*)	$?x : C1 \sqcap \ldots \sqcap Cn$	`(object (is-a C1)(name ?x))`
		`...`
		`(object (is-a Cn)(name ?x))`

(*b*)	$?x : C1 \sqcup \ldots \sqcup Cn$	`(object (is-a C1| ... |Cn)(name ?x))`

(*c*)	$?x : \neg(C1 \sqcap \ldots \sqcap Cn)$	`(object (is-a ~C1)(name ?x))`
		`...`
		`(object (is-a ~Cn)(name ?x))`

(*d*)	$?x : \neg(C1 \sqcup \ldots \sqcup Cn)$	`(object (is-a C1| ... |~Cn)(name ?x))`
(*e*)	$\langle ?x, b \rangle : R$	`(object (name ?x)(R $? b $?))`
(*f*)	$?x : \exists R.C$	`(object (name ?x)(R $?L))` `(test (exists $?L C))`
(*g*)	$?x : \forall R.C$	`(object (name ?x)(R $?L))` `(test (foreach $?L C))`
(*h*)	$?x \geq nR$	`(object (name ?x)(R $?L))` `(test (>= (length$ $?L) n))`

Table 2: Examples of mapping DL constraints on COOL object patterns.

CWA, only existing classes can be defined as class constraints. In other words, we cannot define a class constraint using an undefined class, as OWL allows using anonymous concepts.

4.3.2. Role value restrictions

In section 3.2.2 we state that the attributes of objects should be able to take more than one values in the form of a list. For that reason, all the class attributes of the COOL model are defined as multislots. The simple case of matching objects that have a value *b* in an attribute *R* is depicted in Table 2 (e). In this case, we use the notation `(R $? b $?)` that succeeds if value `b` exists in the value list of the multislot `R` of an object.

Existential and universal constraints can be represented using the two user-defined functions `exists` and `foreach`, which are defined as:

$$\texttt{(exists \$?L C)} \rightarrow TRUE \Leftrightarrow \exists v \in \texttt{L}, v \in Obj(\texttt{C})$$
$$\texttt{(foreach \$?L C)} \rightarrow TRUE \Leftrightarrow |\texttt{L}| = 0 \vee \forall v \in \texttt{L}, v \in Obj(\texttt{C})$$

More specifically, an existential restriction about an ontology property of the form $?x : \exists R.C$ denotes an instance that has at least one instance value in the role *R* that belongs to the concept *C*. This is represented in COOL in terms of objects, using the built-in `test` function of CLIPS for checking the result of Boolean functions in rule bodies, and the function `exists` in order to check for the existence of at least one object of the class `C` in the value list `$?L` (Table 2 (f)).

A universal restriction about an ontology role of the form $?x : \forall R.C$ denotes an instance that has no values for the role *R* or all the values of *R* belong to the concept *C*. This is represented in COOL using the function `foreach` in order to check if the list `$?L` is empty or if all the values of `$?L` are of class type `C` (Table 2 (g)).

Cardinality constraints about role values are checked by counting the number of multislot values. Note that any cardinality constraint is checked following the UNA. For example, if

$\langle$`nick, csd`$\rangle$: `isHeadOf`,
$\langle$`nick, informatics`$\rangle$: `isHeadOf` and
$\langle$`csd, informatics`$\rangle$: `sameAs`

then $\texttt{nick}_o$ would have both the $\texttt{csd}_o$ and $\texttt{informatics}_o$ objects in the $\texttt{isHeadOf}_o$ multislot and thus, $\texttt{nick}_o$ satisfies the object pattern for the constraint $?x : 2\ isHeadOf$, although the `csd` and `informatics` instances are the same (Table 2 (h)).

Finally, more complex constraints can be defined by combining primitive ones. For example, a concept restriction of the form $?x : C1 \sqcap (C2 \sqcup C3)$ can be represented by the object patterns `(object (is-a C1) (name ?x))` and `(object (is-a C2|C3) (name ?x))`.

4.4. Example of Mixing Facts and Object Patterns

We present an example of a rule-based application that combines the fact-based production rules capabilities of CLIPS and the COOL object patterns, based on the results of the CLIPS-OWL library[11]. The OWL ontological knowledge of the example derives from the ontology of Table 3 that was taken from [11] and defines university regulations. More specifically:

[11]The CLIPS-OWL library as well as the OO production rule program of the example can be downloaded from http://lpis.csd.auth.gr/systems/CLIPSOWL/clipsowl.rar.

$u1$. A full professor (FP) is a faculty member (FM)
$u2$. A non-teaching full professor (NFP) is a full professor that does not teach (TC) any course (Co)
$u3$. A course is an advanced (AC) or basic (BC) course
$u4$. The advanced and basic courses are disjoint
$u5$. mary is an instance of full professor and teaches only advanced courses
$u6$. paul is an instance of student (St)
$u7$. john is an instance of full professor
$u8$. Artificial Intelligence (ai) is an advanced course
$u9$. Knowledge representation (kr) is an instance of topic (Tp)
$u10$. Logic programming (lp) is an instance of topic
$u11$. john teaches artificial intelligence

We use the following facts in CLIPS, in accordance to the rule predicates that are defined in the example of [11].

`(mayDoThesis ?x ?y)`. Student `?x` is eligible to do a thesis with professor `?y`.
`(curr ?x ?y)`. Student `?x` has topic `?y` in his/her curriculum.
`(expert ?x ?y)`. Professor `?x` is an expert on topic `?y`.
`(exam ?x ?y)`. Student `?x` passed the exam on topic `?y`.
`(subject ?x ?y)`. Course `?x` covers topic `?y`.

The goal of the rule program is to derive facts of the form `(mayDoThesis x y)` that denote with which professor (`y`) a student (`x`) is eligible to do a thesis, based on the knowledge that is modeled in the ontology. Therefore, we do not use rules in order to alter the ontological knowledge and to represent more complex relationships. Instead, we use the ontological knowledge as the basis for developing a rule-based application, as we stated in our motivation (section 2.5).

The generated COOL model is depicted in Figure 10. Figure 11 depicts the CLIPS production rule version of the backward rule program presented in [11]. More specifically, the `curr` rule asserts a fact `(curr ?x ?z)` if `?x` passed the exam in course `?y` that covers topic `?z`. The

#	OWL Axioms (DL Syntax)
$u1$	$\mathtt{FP} \sqsubseteq \mathtt{FM}$
$u2$	$\mathtt{NFP} \equiv \mathtt{FP} \sqcap \neg\, \mathtt{TC.Co}$
$u3$	$\mathtt{AC} \sqcup \mathtt{BC} \equiv \mathtt{Co}$
$u4$	$\mathtt{AC} \sqcap \mathtt{BC} \equiv \bot$
$u5$	$\mathtt{mary} : \mathtt{FP} \sqcap \forall\, \mathtt{TC.AC}$
$u6$	$\mathtt{paul} : \mathtt{St}$
$u7$	$\mathtt{john} : \mathtt{FP}$
$u8$	$\mathtt{ai} : \mathtt{AC}$
$u9$	$\mathtt{kr} : \mathtt{Tp}$
$u10$	$\mathtt{lp} : \mathtt{Tp}$
$u11$	$\langle\, \mathtt{john}, \mathtt{ai} \,\rangle : \mathtt{TC}$

Table 3: University regulations.

mayDoThesis rule asserts a fact (mayDoThesis ?x ?y) if ?x has topic ?z in the curriculum, ?y is an expert on ?z, and ?y is a faculty member that teaches at least one advanced course. Therefore, by loading the facts

(subject [ai] [kr]),
(subject [ai] [lp]),
(expert [mary] [lp]),
(expert [john] [kr]), and
(exam [paul] [ai]),

and running the production rules of Figure 11, we result in the addition of the fact (mayDoThesis [paul] [john]) in the KB. More specifically, the curr rule is activated and asserts facts (curr [paul] [kr]) and (curr [paul] [lp]). Then, the mayDoThesis rule is activated, since fact (expert [john] [kr]) exists and fact (curr [paul] [kr]) has been added by the curr rule. Note the way object patterns are used in order to restrict the objects to belong to certain classes or to have certain attribute values.

4.5. Memory Consumption and Rule Activation Time

In order to test the efficiency of the OO representation against the trivial fact-based representation of the extensional DL reasoner's semantics, we generated the COOL model and the set of

```
(defclass owl:Thing (is-a USER)
  (multislot TC
   (type INSTANCE-NAME)
   (allowed-classes owl:Thing)))
(defclass FM (is-a owl:Thing))
(defclass CO (is-a owl:Thing))
(defclass AC (is-a CO))
(defclass NFP (is-a owl:Thing))
(defclass FP (is-a FM))
(defclass TP (is-a owl:Thing))
(defclass ST (is-a owl:Thing))
(defclass BC (is-a CO))
```

(a) COOL classes.

```
(make-instance [ai] of AC)
(make-instance [lp] of TP)
(make-instance [john] of FP
  (TC [ai]))
(make-instance [paul] of ST)
(make-instance [kr] of TP)
(make-instance [mary] of FP)
```

(b) COOL instances.

Figure 10: The example COOL model of university regulations.

```
(defrule mayDoThesis
   (object (is-a ST)(name ?x))
   (object (is-a TP)(name ?z))
   (object (is-a FM)(name ?y)
      (TC $?L))
   (test (exists $?L AC))
   (curr ?x ?z)
   (expert ?y ?z)
=>
   (assert (mayDoThesis ?x ?y)))
```

(a) The `maydothesis` rule.

```
(defrule curr
   (object (is-a ST)(name ?x))
   (object (is-a TP)(name ?z))
   (object (is-a CO)(name ?y))
   (exam ?x ?y)
   (subject ?y ?z)
=>
   (assert (curr ?x ?z)))
```

(b) The `curr` rule.

Figure 11: The production rules of the example.

the plain CLIPS facts of the UOBM DL ontology [29] after Pellet DL reasoning. The COOL model was generated by the CLIPS-OWL library, whereas the CLIPS facts were generated by a simple Java tool we developed that traverses Pellet's KB and produces the corresponding facts. Table 4 depicts the memory required by CLIPS to load each model and the number of classes, objects and facts that each model involves.

UOBM is an OWL DL ontology for a university domain. Its TBox consists of 69 concepts and 43 roles. Note that the corresponding COOL model contains more classes than the actual concepts of the UOBM ontology. This happens due to the transformation principles that require to create more OO classes in order to represent multiple instance class membership relationships (see Algorithm 9). For ABox data, the benchmark provides an instance generator tool. In our experiments, we generated a dataset of one university and 20 departments (UOBM-1).

The advantage of the OO representation against the fact-based representation becomes obvi-

	COOL Model	**Fact Model**
CLIPS memory	29 MB	75 MB
Number of classes	646	-
Number of objects	25,460	-
Number of facts	-	517,490

Table 4: Memory requirements and the number of classes, objects and facts.

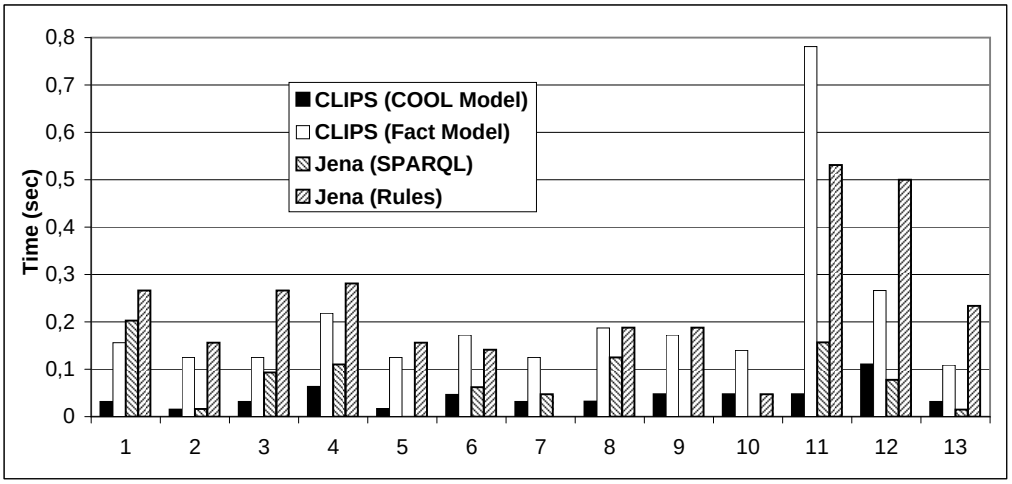


Figure 12: Rule activation times.

ous by the information in Table 4. More specifically, CLIPS-OWL generates a more "machine-friendly" model that requires only 29 MB of main memory to be loaded in CLIPS. On the other hand, the fact model results in 517,490 facts that require more than twice the memory size required by the COOL model.

The UOBM benchmark defines 13 extensional queries, that is, queries about instances and their values. We expressed these queries in terms of COOL and fact constraints in CLIPS production rules and we used these rules in order to test the rule activation performance, that is, to test the constraint checking capabilities of each model in terms of time. Regarding the completeness of the mapping algorithms in terms of preserving the results of DL reasoning procedure, the 13 query rules in CLIPS returned the same set of objects that we obtained by applying the same queries directly in the Pellet reasoner, following the UNA.

Figure 12 depicts the rule activation time of each rule in the two models. In all queries, the OO constraints were checked faster that the corresponding fact-based, showing the effectiveness of the OO representation that encapsulates all the related knowledge about a resource in a single object definition.

To elaborate further on the effectiveness of the OO model, we present in Figure 13 the SPARQL version of the query 11 of the UOBM benchmark and the two corresponding rules that we generated for each model. The query11-fact rule needs almost 0.8 second to be activated (Figure 13c), while the query11-cool rule (Figure 13b) needed less than 0.1 second. This great difference in the activation time lays on the fact that the fact-based rule needs to join 5 triples in its condition, whereas the COOL-based rule needs only a single join among objects. This evidence justifies the efficiency of the OO representation that collects all the related information about a resource in an object and, instead of joining many triples in order to check constraints, we are able to retrieve directly the values exploiting the message-passing mechanism of the underlying OO environment.

At this point we want to mention that a direct comparison of the query response time of the

```
SELECT DISTINCT ?x
WHERE { ?x rdf:type Person .
        ?x like ?y .
        ?z rdf:type Chair .
        ?z isHeadOf University0 .
        ?z like ?y}
```

(a) SPARQL format.

```
(defrule query11-cool
   (object
      (is-a Person)
      (name ?x)
      (like $? ?y $?))
   (object
      (is-a Chair)
      (name ?z)
      (isHeadOf $? [University0] $?)
      (like $? ?y $?))
=>
   (printout t ?x crlf))
```

(b) COOL format.

```
(defrule query11-fact
   (triple ?x rdf:type Person)
   (triple ?x like ?y)
   (triple ?z rdf:type Chair)
   (triple ?z isHeadOf University0)
   (triple ?z like ?y))
=>
   (printout t ?x crlf))
```

(c) Fact format.

Figure 13: Different representations of the query 11 of UOBM.

CLIPS engine with other engines, such as Jena [2] or Jess[12], is out of the scope of this paper. The purpose of the experimental evaluation of CLIPS through the CLIPS-OWL library is (a) to test the soundness and the ability of CLIPS-OWL, and consequently of our OO mapping methodology, to preserve the inferencing results that are produced by the Pellet DL reasoner, and (b) to testify the effectiveness of the OO model against the fact-based model during queries. Therefore, our intention is to investigate the improvement we achieve in rule activation, and consequently in rule execution, following the OO and fact representations in the same OO rule engine. However, for the sake of completeness, we depict also in Figure 12 the query response and rule activation times of Jena over the inferencing model that is generated by the Pellet DL reasoner for the UOBM ontology. More specifically, we used the Pellet API for Jena in order

[12]http://herzberg.ca.sandia.gov/

to generate a Jena model for the tested ontology, and we applied the 13 extensional queries in the form of SPARQL queries and Jena rules. In general, the SPARQL engine of Jena performs better than Jena rules. However, the incorporation of the ontology semantics into rule programs can be done via rules only, and therefore, the Jena rules are more practical. Moreover, the fact-based CLIPS representation can, in general, provide faster rule activation than of using Jena rules and slower querying answering than of SPARQL queries. However, the results showed that the COOL model performs better than the SPARQL engine.

5. Related Work

The similarities of OWL and OO modeling have attracted the interested of many research proposals. OntoJava [30] converts RDF Schema and RuleML into Java classes. In that way, every RDF class is mapped on a Java class, every RDF property on a class attribute and every RDF instance on a Java object. However, this mapping is not based on an RDF reasoner, but it rather tries to capture semantics using only OO principles. Therefore, OntoJava is able to preserve only a portion of RDF semantics, unable to perform, for example, sophisticated instance classification. In our work, we define a richer mapping of OWL semantics on the OO model. However, we do not use OO principles to perform OWL reasoning, like OntoJava does for RDF reasoning, but we use the OO model to represent OWL semantics after DL reasoning.

In [31] an approach is presented for mapping OWL on Java. Like OntoJava, it does not use a DL reasoner to infer the semantic relationships, but it captures them using directly OO principles. To this end, some OWL semantics are not handled, such as the inverse functional properties. Furthermore, the approach does not define a mapping of OWL instances into Java objects, but it deals only with concepts and roles. This is a limitation, since several OWL semantics are applied on instances (ABox), such as same individuals (`owl:sameAs`).

There are other similar approaches to the above, such as the RDFReactor [32]. The major difference of such approaches to ours is that their goal is to enable the use of the ontological knowledge in OO programming, for example in Java, without being interested in preserving the complete semantics of OWL. In order to preserve these semantics, the OO principles are not enough and a reasoner is needed (see section 2.4). Our intention is to preserve the extensional OWL semantics through a specially defined OO model and to use this model in OO rule engines in order to answer instance related constraints. For that reason, we are based on the inferencing capabilities of DL reasoners, mapping their KBs on the OO model.

In [33] a framework is presented for combining the OO and ontological representations. The idea is to develop programs that are expressed both in terms of Java classes and OWL concepts. This approach does not define any mapping of OWL on the OO model but it introduces an interesting combination of the two knowledge representation paradigms.

ActiveRDF [34] is an RDF mapping approach to the OO model in Ruby. The purpose of this mapping is to generate an API in order to manipulate and query RDF ontologies. The scripting-nature of Ruby allows for the dynamic modification of classes and attributes. However, the high expressivity of OWL requires a more sophisticated mapping than RDF, that we achieve by utilizing a DL reasoner in order to capture OWL semantics. We are not interested in the dynamic modification of the generated OO model, since our intention is not to develop a programming API. We use the OO model only as query infrastructure, answering constraints.

In [35], disjunctive logic programming is extended with OO constructs, implementing an OO environment on top of DLV. This is a different application domain to ours, where we define a mapping procedure of OWL ontologies on the OO model using a DL reasoner.

SWCLOS [36] is an OWL reasoner developed on top of the Common Lisp Object System (CLOS) that allows LISP programmers to develop OO systems. SWCLOS maps ontologies on the OO model of CLOS and performs inferencing using entailments rules [37] and not a DL reasoner. Therefore, the mapping procedure of SWCLOS does not capture the complete set of OWL semantics. For example, the instance class membership semantics of equivalent concepts is not treated by the model. Moreover, this approach is actually a homogenous one, since any rule program coexists with the entailment rules in the same RB. Therefore, SWCLOS targets at different application environments, as we have described in section 2.5.

In [38], we describe O-DEVICE, our effort to build an OWL reasoner following the OO capabilities of CLIPS. The implementation is characterized by the application of entailment rules for OWL reasoning, and thus, it is a homogeneous system targeting mainly at OWL reasoning like SWCLOS. Although O-DEVICE is defined on top of CLIPS, the algorithms of generating the OO model, as well the OO model itself, are different to the ones that are described in this work. This happens since the OO model in O-DEVICE is created taking into consideration the entailment rules that should be applied in order to perform OWL reasoning in CLIPS. In the present work, OWL semantics derive directly from the DL reasoner and therefore, we are only interested in defining the mapping algorithms of its KB on the OO model. For example, O-DEVICE requires the definition of meta-objects, which are special objects necessary to store concept and role information. For that reason, O-DEVICE requires the loading of an RDF and OWL meta-model. This approach is also followed by SWCLOS. Since OWL reasoning in our OO Unidirectional Homogeneous framework is performed by a DL reasoner, there is no need for such extra meta-models and meta-objects, resulting in a simpler OO model. Recall also that the OWL reasoning with entailments is not complete. The differences of our approach to homogeneous ones are described in sections 2.3.2 and 2.5.

Examples of hybrid unidirectional approaches are the [11][18][13][39]. AL-log [11] is a combination of ALC DL and positive Datalog, where only concepts can be used as constraints in rule bodies. A query is examined by using backward chaining and a DL reasoner in order to prove the DL constraints at runtime. DL-log [18] combines disjunctive Datalog with ALC, extending AL-log to property constraints in rule bodies. In [13], XSB is interfaced with Pellet and only DL constraints of the form $X : C$ are used in the rule bodies that are checked by the reasoner. In [39], a hybrid framework combines a rule language with a constraint language, following the typical hybrid architecture. In section 2.5.1 we explained that these approaches introduce an unnecessary communication overhead between the rule engine and the DL reasoner in isolated back-end ontological environments. For that reason, we map the DL reasoner KB on the rule engine and the OO model is used to check instance constraints. The constraints are answered following the rule paradigm (UNA, CWA) and we are able to represent both instance class membership and role values constraints.

SWRLJessTab [22] and DL-Florid [16] are examples of hybrid bidirectional approaches. SWRLJessTab maps SWRL rules to Jess rules and OWL entailed facts from the Racer reasoner to Jess facts. Jess rules operate over Jess's KB and the inferred information is translated back to Racer in order to perform DL reasoning again. If we ignore the bridge from Jess to Racer, we result in a Unidirectional Homogeneous framework similar to ours. However, SWRLJessTab performs a trivial translation of OWL axioms into facts, while we employ OO principles (see section 3). DL-Florid [16] is a combination of DL and F-Logic [40]. The DL symbols do not need to be disjoint with the rule part and thus, concept memberships and role instances may be derived. There are major differences between our OO approach and frame-based approaches, such as the DL-Florid and F-OWL [41]. In these systems, F-Logic is used as an ontology representation

language only, following a frame-based syntax. The knowledge is finally translated into facts: F-Logic facts in DL-Florid and XSB facts in F-OWL. On the other hand, our OO approach is based totally on OO principles, using the OO model for both expressing and physically storing ontology semantics.

Protege[13] is a free, open source ontology editor and knowledge-base framework written in Java that gives the opportunity to save (export) the developed ontology as CLIPS code using OO constructs. However, the purpose of this transformation is not to define an OO model that encapsulates the extensional knowledge of the ontology, but to store the ontology information in the form of simple meta-objects. For that reason, it does not utilize any OWL inference mechanism. To the best of our knowledge, our work is the first one that defines the way OO modeling principles can be used in order to store/index the complete ABox knowledge of DL reasoners.

Note that our approach, although it utilizes a rule engine over ontological knowledge, has different application domains to the approaches that perform a *tight integration of rules and ontologies*, such as KAON2 [42], SWRL [43] and other homogeneous-oriented systems. The goal of such approaches is to enable the efficient and decidable integration of rules and ontologies in order to express richer ontological relations. Our intention is not to manipulate ontologies with rules. Instead, we use the ontological knowledge after DL reasoning inside rule-based applications, as a constraint model, where the ontology constraints exist only in rule bodies.

This work is a revised and extended version of our short paper in [44], describing in detail the ontology mapping algorithm. More specifically, we present here a prototype implementation of our OO methodology (CLIPS-OWL), we describe how the fact-based rule paradigm can be mixed with the OO rule paradigm in CLIPS over an ontological vocabulary, and we present experimental results.

6. Conclusions and Future Work

There are different approaches that allow the already existing and well-known infrastructure of rule engines to gain access in the OWL ontological representation of information. We argue that there is not a single approach suitable for every problem, and the suitability depends on the application domain and user requirements.

In this paper, we target at domains where there is an isolated back-end ontological model, such the ones we have mentioned in the introduction. Our approach exploits the knowledge representation power of OO rule engines and defines a methodology for mapping the extensional OWL ontological knowledge, which is inferred by DL reasoners, on the OO model. The purpose of this mapping is to allow this knowledge to be imported in OO rule engines as an OO model, giving the opportunity to treat instance-related ontological constraints as OO queries. In that way, (a) we eliminate runtime communication overheads between rule engines and external OWL reasoning components for checking instance constraints, (b) there is no need to modify the architecture of the DL reasoner and the rule engine, (c) we exploit the inferencing capabilities of the DL reasoning paradigm, and (d) we achieve a more compact and efficient representation of ontology semantics in the rule engine, than having to explicitly store them in the form of facts, resulting in better response time of instance related queries.

[13] http://protege.stanford.edu/

We presented also CLIPS-OWL, a prototype implementation of our mapping algorithms using the COOL OO language of the CLIPS production rule engine and the Pellet DL reasoner to handle OWL semantics. We elaborated also on experiments we performed in order to justify the advantages of the OO representation against the fact-based representation.

Currently, we are implementing our mapping methodology in other OO rule engines, such as the forward-chaining inference rule engine of Drools that allows the existence of a Java OO model (beans). Bear in mind that our mapping methodology can be applied in any OO environment, including Java, as we have mentioned in the introduction. Moreover, we are using our framework in the domain of Software AntiPatterns [45], which represent software project management knowledge. The idea is to use an ontology-based representation of AntiPatterns and to build rule-based systems on top of them in order to help project managers during decision making. This is also an example of an isolated back-end ontological model where the rule program is used to draw conclusions over ontological knowledge. We plan also to release a plugin for Protege, extending its current module for creating CLIPS code, and to implement the DIG interface in order to support any DIG-compliant DL reasoner.

References

[1] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003.

[2] B. McBride, Jena: Implementing the rdf model and syntax specification, in: SemWeb, 2001.

[3] S. Bechhofer, R. Möller, P. Crowther, The dig description logic interface, in: D. Calvanese, G. De Giacomo, E. Franconi (Eds.), Description Logics, Vol. 81 of CEUR Workshop Proceedings, CEUR-WS, 2003.

[4] E. Oren, A. Haller, C. Mesnage, M. Hauswirth, B. Heitmann, S. Decker, A flexible integration framework for semantic web 2.0 applications, IEEE Softw. 24 (5) (2007) 64–71.

[5] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical owl-dl reasoner, Web Semantics: Science, Services and Agents on the World Wide Web 5 (2) (2007) 51–53.

[6] G. Antoniou, F. van Harmelen, A Semantic Web Primer (Cooperative Information Systems), The MIT Press, 2004.

[7] V. Haarslev, R. Möller, Racer: A core inference engine for the semantic web, in: Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003), Sanibel Island, Florida, USA, 2003, pp. 27–36.

[8] D. Tsarkov, I. Horrocks, Fact++ description logic reasoner: System description, in: Third International Joint Conference on Automated Reasoning (IJCAR), Vol. 4130 LNAI, 2006, pp. 292–297.

[9] F. Baader, U. Sattler, An overview of tableau algorithms for description logics, Studia Logica 69 (2001) 5–40.

[10] G. Antoniou, C. V. Damasio, B. Grosof, I. Horrocks, M. Kifer, J. Maluszynski, Peter, Combining rules and ontologies. a survey., deliverable REWERSE-DEL-2005-I3-D3, Institutionen for datavetenskap, Linkopings Universitetet (2005).

[11] F. M. Donini, M. Lenzerini, D. Nardi, A. Schaerf, Al-log: integrating datalog and description logics, J. of Intelligent and Cooperative Information Systems 10 (1998) 227–252.

[12] R. Rosati, Towards expressive kr systems integrating datalog and description logics: preliminary report, in: Description Logics, 1999, pp. 160–164.

[13] W. Drabent, J. Henriksson, J. Maluszynski, Hd-rules: A hybrid system interfacing prolog with dl-reasoners., in: A. Polleres, D. Pearce, S. Heymans, E. Ruckhaus (Eds.), ALPSWS, Vol. 287 of CEUR Workshop Proceedings, CEUR-WS.org, 2007.

[14] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, H. Tompits, Combining answer set programming with description logics for the semantic web, Artif. Intell. 172 (12-13) (2008) 1495–1539.

[15] R. Rosati, Semantic and computational advantages of the safe integration of ontologies and rules, in: PPSWR-05, Vol. 3703 of LNCS, 2005, pp. 50–64.

[16] H. Kattenstroth, W. May, F. Schenk, Combining owl with f-logic rules and defaults., in: A. Polleres, D. Pearce, S. Heymans, E. Ruckhaus (Eds.), ALPSWS, Vol. 287 of CEUR Workshop Proceedings, CEUR-WS.org, 2007.

[17] K. Wang, D. Billington, J. Blee, G. Antoniou, Combining description logic and defeasible logic for the semantic web., in: Rules and Rule Markup Languages for the Semantic Web, Springer-Verlag, 2004, pp. 170–181.

[18] R. Rosati, Dl+log: Tight integration of description logics and disjunctive datalog, in: KR, 2006, pp. 68–78.

[19] S. Heymans, J. D. bruijn, L. Predoiu, C. Feier, D. V. niewenborgh, Guarded hybrid knowledge bases, Theory Pract. Log. Program. 8 (3) (2008) 411–429.
[20] J. Mei, Z. Lin, H. Boley, $\mathcal{ALC}_p^u$: An integration of description logic and general rules, in: Web Reasoning and Rule Systems (RR), 2007, pp. 163–177.
[21] B. Grosof, I. Horrocks, R. Volz, S. Decker, Description Logic Programs: Combining Logic Programs with Description Logics, in: Proc. of WWW-2003, Budapest, Hungary, 2003.
[22] C. Golbreich, A. Imai, Combining swrl rules and owl ontologies with protege owl plugin, jess, and racer, in: 7th International Protege Conference, 2004.
[23] J. Martin, Principles of object-oriented analysis and design, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
[24] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon, Common lisp object system specification, SIGPLAN Not. 23 (SI) (1988) 1–142.
[25] G. Riley, Clips: An expert system building tool, in: Proceedings of the Technology 2001Conference, 1991.
[26] S. Koide, J. Aasman, S. Haflich, Owl vs. object oriented programming, in: Workshop on Semantic Web Enabled Software Engineering (SWESE), 2005, galway, Ireland.
[27] J. de Bruijn, R. Lara, A. Polleres, D. Fensel, Owl dl vs. owl flight: conceptual modeling and reasoning for the semantic web, in: WWW '05: Proceedings of the 14th international conference on World Wide Web, ACM Press, New York, NY, USA, 2005, pp. 623–632.
[28] P. F. Patel-Schneider, I. Horrocks, A comparison of two modelling paradigms in the semantic web, Web Semant. 5 (4) (2007) 240–250.
[29] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, S. Liu, Towards a complete owl ontology benchmark, in: European Semantic Web Conference, 2006, pp. 125–139.
[30] A. Eberhart, Automatic generation of java/sql based inference engines from rdf schema and ruleml, in: ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web, Springer-Verlag, London, UK, 2002, pp. 102–116.
[31] A. Kalyanpur, D. J. Pastor, S. Battle, J. A. Padget, Automatic mapping of owl ontologies into java., in: F. Maurer, G. Ruhe (Eds.), SEKE, 2004, pp. 98–103.
[32] M. Volkel, Y. Sure, Rdfreactor - from ontologies to programmatic data access, Poster and Demo at International Semantic Web Conference (ISWC) 2005, Galway, Ireland (2005).
[33] C. Puleston, B. Parsia, J. Cunningham, A. Rector, Integrating object-oriented and ontological representations: A case study in java and owl, in: International Semantic Web Conference (ISWC), 2009.
[34] E. Oren, R. Delbru, S. Gerke, A. Haller, S. Decker, Activerdf: object-oriented semantic web programming, in: WWW '07: Proceedings of the 16th international conference on World Wide Web, ACM, 2007, pp. 817–824.
[35] A. W. Systeme, F. Ricca, F. Ricca, N. Leone, N. Leone, Disjunctive logic programming with types and objects: The dlv+ system, Journal of Applied Logic, Elsevier, Volume 5, Issue 3 (2007) 545–573.
[36] S. Koide, H. Takeda, Owl-full reasoning from an object oriented perspective, in: Asian Semantic Web Conference (ASWC), 2006, pp. 263–277.
[37] H. J. ter Horst, Completeness, decidability and complexity of entailment for rdf schema and a semantic extension involving the owl vocabulary, Web Semantics: Science, Services and Agents on the World Wide Web 3 (2-3) (2005) 79–115.
[38] G. Meditskos, N. Bassiliades, A rule-based object-oriented owl reasoner, IEEE Trans. on Knowl. and Data Eng. 20 (3) (2008) 397–410.
[39] J. Henriksson, T. U. Dresden, Combining safe rules and ontologies by interfacing of reasoners, in: In PPSWR 2006, number 4187 in LNCS, Springer, 2006, pp. 33–47.
[40] M. Kifer, G. Lausen, J. Wu, Logical foundations of object-oriented and frame-based languages, Journal of ACM 42 (1995) 741–843.
[41] Y. Zou, T. W. Finin, H. Chen, F-owl: An inference engine for semantic web, in: Formal Approaches to Agent-Based Systems, 2004, pp. 238–248.
[42] U. Hustadt, B. Motik, U. Sattler, Reducing shiq description logic to disjunctive datalog programs, in: Proc. of the 9th International Conference on Knowledge Representation and Reasoning (KR2004), 2004, pp. 152–162.
[43] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, Swrl: A semantic web rule language combining owl and ruleml, Tech. rep., W3C Member Submission (2004).
URL `http://www.w3.org/Submission/SWRL/`
[44] G. Meditskos, N. Bassiliades, Hoopo: A hybrid object-oriented integration of production rules owl ontologies, in: ECAI 2008 - 18th European Conference on Artificial Intelligence, IOS Press, 2008, pp. 729–730.
[45] D. Settas, I. Stamelos, Towards a dynamic ontology based software project management antipattern intelligent system, in: 19th IEEE International Conference on Tools with Artificial Intelligence - Vol.1 (ICTAI 2007), IEEE Computer Society, 2007, pp. 186–193.