A Rule-based Object-Oriented OWL Reasoner

Georgios Meditskos and Nick Bassiliades

Abstract— In this paper we describe O-DEVICE, a memory-based knowledge base system for reasoning and querying OWL ontologies by implementing RDF/OWL entailments in the form of production rules in order to apply the formal semantics of the language. Our approach is based on a transformation procedure of OWL ontologies into an Object-Oriented schema and the application of inference production rules over the generated objects in order to implement the various semantics of OWL. In order to enhance the performance of the system, we introduce a dynamic approach of generating production rules for ABOX reasoning and an incremental approach of loading ontologies. O-DEVICE is built over the CLIPS production rule system, using the object-oriented language COOL to model and handle ontology concepts and RDF resources. One of the contributions of our work is that we enable a well-known and efficient production rule system to handle OWL ontologies. We argue that although native OWL rule reasoners may process ontology information faster, they lack some of the key features that rule systems offer, such as the efficient manipulation of the information through complex rule programs. We present a comparison of our system with other OWL reasoners, showing that O-DEVICE can constitute a practical rule environment for ontology manipulation.

Index Terms— Inference engines, Object-Oriented Programming, Ontology languages, Rule-based processing.

1 Introduction

To exploit the Web to its full extend, information should become understandable not only to humans but to machines too. Today's Web is targeted at humans, making the discovery of information a time consuming task. Search engines need the ability to semantically understand and exploit the available knowledge, without relying on the syntax of information. Moreover, Web is continuously enriched with services. In such a service-oriented architecture (SOA) each service can communicate with others by passing messages and services can be composed into more complex ones. In order to enable automated service discovery and composition, two fundamental issues for the successful proliferation of SOAs, services should be well-described.

The Semantic Web initiative [1] tries to solve such problems by suggesting standards, tools and languages for information annotation. Ontologies play a key role to the evolution of the Semantic Web and are widely used to represent knowledge by describing data in a formal way. OWL [2] 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. Thus, ontology reasoning systems appear to be of great importance.

Existing implementations of OWL reasoners are based on several approaches. The Description Logic reasoners (e.g. Pellet [3], RacerPro [4]) implement tableaux algorithms [5], exploiting the research that has been done on algorithms for the description logics knowledge representation formalism on which OWL is based. Datalog-driven engines (e.g. KAON2 [6]) reduce a SHIQ(D) KB to a disjunctive datalog program [7]. Rule-based reasoners (e.g. OWLIM [8], OWLJessKB [9]) use a rule engine to define rules for inferencing. F-Logic [10] based engines (e.g. F-OWL [11], Ontobroker [12]) use frames in order to manipulate the ontology information. FOL theorem provers (e.g. Vampire [13]) translate DL axioms into a FO theory by mapping DL concepts and roles names into unary and binary predicates. Finally, reasoners based on conceptual graphs (e.g. Corese [14]) transform ontology information into a conceptual graph formalism.

In this paper we present O-DEVICE, a production rule-based system for inferencing about and querying OWL ontologies. We use CLIPS [15], a well-known production rule engine, and we augment it with an OWL-to-objects mapping mechanism in order to handle OWL semantics following an object-oriented (here after OO) approach. Our implementation handles the OWL Lite sublanguage, offering also support for some OWL DL constructs, such as partial union of classes, the `owl:hasValue` construct and class disjointness. Currently, we do not support class definitions by instance enumeration (`owl:oneOf`), complement classes and data ranges.

The work presented here is based on the experience gained by previous efforts ([16], [17]) to develop a rule reasoning system using static production rules. However, in a memory-based rule inference system, like O-DEVICE, memory utilization is very important and the efficiency depends mainly on the *quantity* and the *quality* of rules. *Quantity* refers to the number of implemented entailments. The more they are, the more semantics can be handled and thus the more complete the system is. But the number of inference rules affects performance. Our approach targets at developing a practical reasoning system, able to perform with reasonable (time and memory) requirements. *Quality* refers to the implementation aspects of these rules. The semantics of each OWL construct can be handled by rules implemented in different ways. To this end, we have followed a *dynamic rule generation* method that is able to handle larger number of objects by

• *The authors are with Department of Informatics, Aristotle University of Thessaloniki, 54124, Greece. E-mail: {gmeditsk, nbassili}@csd.auth.gr.*

xxxx-xxxx/0x/$xx.00 © 200x IEEE
Digital Object Indentifier 10.1109/TKDE.2007.190699 1041-4347/$25.00 © 2007 IEEE

restricting the search space where they are applied. The *domain-dependent* rules we generate have simpler conditions and thus faster activation time than the corresponding static/generic ones. Other improvements include the incremental loading of triples and the cyclic, partial application of the production rules. Both mechanisms aim to reduce the size of the RETE network that needs to be built in order to match objects to rules, leading not only to memory saving but also to speed-up due to less memory management activities. All the above are testified via experimental results that also compare our system to others.

The rest of the paper is organized as follows: in section 2 we describe our motivation for using an existing production rule system and for transforming OWL to an OO schema. In section 3 we give a short background of the CLIPS rule engine. In section 4 we give a detailed description of the ontology mapping and inference rules. In section 5 we analyze the loading procedure. In section 6 we describe the deductive query language of the system while in section 7 we present experimental results. Finally, in sections 8 and 9 we present related work and conclude giving future direction, respectively.

2 The Motivation

Our motivation is to combine OWL ontologies, the rule programming paradigm and the OO model using a well-known and efficient rule inference engine in order to enable it to handle OWL ontologies in a practical manner.

Several reasoners offer the possibility of connecting them to external applications, e.g. through the DIG [18] interface, as most DL reasoners and KAON2 supports, justifying the need of further exploitation of the reasoning results. Our approach targets at the exploitation of the OWL ontological information via a *rule engine*.

In existing reasoning implementations, although it is possible to manipulate ontologies using a rule notation, e.g. SWRL [19] language in KAON2, or to perform queries over the ontology, e.g. SPARQL [20], it is not possible (or it is not efficient at least) to define a complete rule program over the ontology since they are not dedicated rule engines. To this end, the use of a rule system able to reason over ontologies, gives the opportunity to utilize directly the ontology information by building knowledge-based systems. Ontologies can be inserted into the system and, after the materialization of the semantics through the reasoning procedure, i.e. the application of inference rules in order to deduce new information, user-defined rules can operate over the materialized knowledge.

However, although *rule-based* OWL reasoners built from scratch, such as OWLIM, may process ontology information and answer single queries fast, since they are optimized for this domain, we argue that they cannot handle complex and large rule programs as efficient as a native rule engine, such as CLIPS.

The motivation behind our OO representation of OWL is twofold. Firstly, the OO form of the information lays 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. By transforming ontologies into the OO model, we enable the implementation of OO rule programs by users, taking also into account that OWL uses classes, properties and objects as well for the definition of concepts for a domain.

Secondly, the transformation enables us to exploit basic features that an OO environment can offer. The native mechanism of COOL for subclass relationships supports class subsumption and transitivity, treating both single and multiple inheritance issues, saving us from the complex and costing procedure of handling hierarchical class relationships and derived consequences, such as class membership and property inheritance.

Moreover, the OO ontology representation can be considered as a trivial but efficient form of indexing. Class definitions embed their properties and property values are encapsulated inside the resource objects, enabling the direct access of property values of a particular object. Every reference to an object's property is handled by system pointers to the corresponding values that are created during the object initialization by CLIPS. In that way, we can directly access property values through the native COOL message passing mechanism.

At this point, two things are worth mentioning. Firstly, the OO model is not able to capture the complete semantics of the OWL language. With the transformation procedure, we want to capture as many semantics as possible, such as class and property inheritance issues. More complex class and property semantics, such as intersection of classes or property transitivity cannot be modeled directly by the OO model and for that reason we implement entailment rules that we present in the following sections. Secondly, since we use a rule engine, we are in a closed-world. However, the mapping mechanism exhibits a dynamic behavior in order to cope with the open-world nature of OWL. Therefore, already created classes may need to change their definition or objects may need to change their type at runtime due to the open-world semantics of OWL. Furthermore, the close-world assumption that rule systems follow during querying is not always an undesirable feature. This depends on the domain of the application and the queries that are to be answered, e.g. queries about negative information [21].

3 The CLIPS Rule Engine

CLIPS [15] supports three different programming paradigms: procedural, rule-based and OO. The semantics of CLIPS production rules 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.

The OO module of CLIPS supports abstraction, inheritance, encapsulation, polymorphism and dynamic binding, integrating procedural, OO and rule-based programming, since classes, properties and objects can be manipulated via rules. The definition of an OO model in CLIPS is performed via the COOL [15] language which

TABLE 1
The Entailment Rules Supported by O-DEVICE

	IF	THEN
rdfs2	p domain d, s p o	s type d
rdfs3	p range r, s p o	o type r
rdfs4a	u p w	u type Resource
rdfs4b	u p w	w type Resource
rdfs7x	p subPropertyOf q, u p w	u q w
rdfs9	u subClassOf w, s type u	s type w
rdfs11	u subClassOf w, w subClassOf t	u subClassOf t
rdfp1	p type FunctionalProperty, u p w, u p o	w sameAs o
rdfp2	p type InverseFunctionalProperty, u p w, s p w	u sameAs s
rdfp3	p type SymmetricProperty, u p w	w p u
rdfp4	p type TransitiveProperty, u p w, w p o	u p o
rdfp8ax	p inverseOf q, u p w	w q u
rdfp8bx	p inverseOf q, u q w	w p u
rdfp11	u p w, u sameAs u', w sameAs w'	u' p w'
rdfp12a	u equivalentClass w	u subClassOf w
rdfp12b	u equivalentClass w	w subClassOf u
rdfp13a	u equivalentProperty w	u subPropertyOf w
rdfp13b	u equivalentProperty w	w subPropertyOf u
rdfp15	u someValuesFrom w, u onProperty p, s p x, x type w	s type u
rdfp16	u allValuesFrom w, u onProperty p, s type u, s p x	x type w

provides the necessary means for defining classes, properties and objects as well as relationships among them.

- **Subclass relationships:** COOL allows the definition of single and multiple class inheritance. This feature is used in our methodology for implementing complex class constructors (section 4.2).
- **Property inheritance:** Properties (in CLIPS they are called slots/multislots) are inherited to subclasses.
- **Object relationships:** Object referencing slots are used to define OWL instance relations.

We exploit these basic OO features of CLIPS in the domain of OWL ontologies through the mapping procedure we describe next in order to treat some of the OWL semantics using the underlying OO environment.

4 Ontology Mapping and Inference Rules

O-DEVICE implements a number of entailments that are presented in Table 1 [22]. Fig. 1 illustrates in the form of an OO logic-like syntax the rules of O-DEVICE, since an exact and detailed presentation of the CLIPS production rules would be an unnecessary complication. However, in the remainder of the paper we present some rule examples in the native CLIPS/COOL syntax in order give a feeling of the implementation. For the logic-like representation we assume that *Tr* is the set of ontology and instance triples, *Cl* is the set of user classes, *Obj* is the set of existing objects and *ext*(*C*) is the extension of class *C*, with $\forall C \in Cl$, $ext(C) \subseteq Obj$. Notice that (a) the set *Obj* equals to the union of all class extensions $Obj = \cup ext(C)$, (b) the set *Obj* includes all OWL instances *Obj*=*ext*(*owl*:*Thing*), and (c) the set *Cl* includes all OWL classes *Cl*=*ext*(*owl*:*Class*). Furthermore, *class*(*o*) is a function that returns the class of the object *o* and *slots*(*C*) returns the set of slots of class *C*. Finally, the expression *t.s* delivers the subject of the triple, *t.o* the object and *t.p* the predicate. In general, the expression *o.s* returns the values of the slot *s* of object *o*.

4.1 Ontology Mapping

The mapping of basic OWL primitives into OO constructs is straightforward: there are classes with properties that model a concept of a particular domain and instances are defined upon them, creating the actual KB by specifying relationships among them. In that way, each OWL class is mapped into a COOL class, each OWL property into a slot (actually, a multislot) of a COOL class and each OWL instance into a COOL object.

4.1.1 Implementing Basic OWL Axioms

The OO schema is implemented in a way so to reflect OWL axioms. We present four basic axioms that characterize our implementation.

- *Axiom 1: Each class is a direct or indirect subclass of the* `owl:Thing` *class*. Therefore, the $C \sqsubseteq$ *owl:Thing* assertion is always satisfiable for every class *C* of the KB.
- *Axiom 2: Every object belongs directly or indirectly to the* `owl:Thing` *class*. Therefore, the *owl:Thing*(*i*) assertion is always satisfiable for every instance *i* of the KB.
- *Axiom 3: Every role P for which no domain class is defined, the system assumes* $\top \sqsubseteq \forall P.owl:Thing$ *and the role is mapped as a slot in the* `owl:Thing` *class*. Therefore, every object inherits the property *P*.
- *Axiom 4: Every role P for which no range constraint is defined, the system assumes* $\top \sqsubseteq \forall P.owl:Thing$. Therefore, *P* can take any value.

In an OWL ontology, classes and properties are defined as instances of appropriate built-in classes, e.g. `owl:Class` or `owl:ObjectProperty`. The system creates the objects that correspond to these instances, which we call *meta-objects*. In that way, properties are still first class citizens, as in RDF and OWL, since they are objects (meta-objects) of the corresponding classes. In order for the system to be able to create the meta-objects, we have predefined the built-in classes and properties of OWL in the form of an OO schema based on the RDF schema of OWL, as it is defined in [23]. The OO implementation of the RDF Schema has been taken from [24].

4.1.2 Transformation Rules

In this section we analyze the transformation rules by presenting also the role of each one during the transformation of the ontology of Fig. 2 in the COOL OO model.

- ***r1***: *Materializes the classes of the OO model.* Each concept *C* of an ontology is mapped into a *defclass* construct, the native construct for defining classes in COOL. This rule is responsible for generating the *Person defclass* construct from the first triple of the example. The *Axiom 1* is used in order to define the class as a subclass of the `owl:Thing`. Notice that at this point, the class has not any slots yet.
- ***r2***: *Generates the attributes of each class of the OO model*. Each axiom $\top \sqsubseteq \forall P.C$ (the domain of *P* is the class *C*) is mapped into a multislot with name *P* in the domain class *C* which should be materialized ($C \in Cl$). By this rule, the

r1: $\forall t \in Tr,\ t.p=rdf:type \wedge t.o=owl:Class \rightarrow t.s \in Cl$
r2: $\forall t \in Tr,\ t.p=rdfs:domain \wedge t.o \in Cl \rightarrow t.s \in slots(t.o)$
r3: $\forall t \in Tr,\ t.p=rdfs:range \wedge t.s \in ext(owl:ObjectProperty) \wedge t.o \in Cl \rightarrow \exists C \in Cl,\ t.s \in slots(C) \wedge C.(t.s).type = INSTANCE\text{-}NAME$
r4: $\forall t \in Tr,\ t.p=rdfs:range \wedge t.s \in ext(owl:DatatypeProperty) \rightarrow \exists C \in Cl,\ t.s \in slots(C) \wedge C.(t.s).type = mapDT(t.o)$
r5: $\forall t \in Tr,\ t.p=rdfs:subClassOf \wedge t.o \in Cl \wedge t.s \in Cl \rightarrow (t.s).is\text{-}a = t.o$
r6: $\forall t \in Tr,\ t.o \in Cl \wedge t.s \notin Obj \wedge t.p=rdf:type \rightarrow t.s \in ext(t.o)$
r7: $\forall t \in Tr,\ t.o \in Cl \wedge t.s \in Obj \wedge t.p=rdf:type \wedge t.o \neq class(t.s) \rightarrow t.s \in ext(t.o \sqcap class(t.s))$
r8: $\forall t \in Tr,\ t.o \in Obj \wedge t.s \in Obj \wedge t.p \in slots(class(t.s)) \wedge t.p \in ext(owl:ObjectProperty) \rightarrow t.o \in (t.s).(t.p)$
r9: $\forall t \in Tr,\ t.o \in Obj \wedge t.s \in Obj \wedge t.p \notin slots(class(t.s)) \wedge t.p \in ext(owl:ObjectProperty) \rightarrow t.s \in ext((t.p).rdfs:domain \sqcap class(t.s))$
r10: $\forall t \in Tr,\ t.p \in ext(owl:ObjectProperty) \wedge (t.p).rdfs:range \in Cl \rightarrow t.o \in ext((t.p).rdfs:range)$
r11: $\forall b,c \in Cl,\ c \in b.owl:disjointWith \wedge (b \sqsubseteq c \vee c \sqsubseteq b) \rightarrow \square$
r12: $\forall p_1, p_2 \in ext(owl:ObjectProperty),\ p_2 \in p_1.owl:inverseOf \rightarrow p_1.rdfs:range = p_2.rdfs:domain \wedge p_1.rdfs:domain = p_2.rdfs:range$
r13: $\forall p_1, p_2 \in ext(rdf:Property),\ p_2 \in p_1.rdfs:subPropertyOf \rightarrow p_1.rdfs:domain \sqsubseteq p_2.rdfs:domain$
r14: $\forall p_1, p_2 \in ext(owl:ObjectProperty),\ p_2 \in p_1.rdfs:subPropertyOf \rightarrow p_1.rdfs:range \subseteq p_2.rdfs:range$
r15: $\forall p_1 \in ext(owl:FunctionalProperty),\ p_2 \in p_1.owl:inverseOf \rightarrow p_2 \in ext(owl:InverseFunctionalProperty)$
r16: $\forall p_1 \in ext(owl:InverseFunctionalProperty),\ p_2 \in p_1.owl:\ inverseOf \rightarrow p_2 \in ext(owl:\ FunctionalProperty)$
r17: $\forall p \in ext(owl:TransitiveProperty), \forall x \in ext(p.rdfs:domain),\ \forall o \in x.p,\ (o.p\text{-}x.p) \neq \varnothing \rightarrow (o.p\text{-}x.p) \subseteq x.p$
r18: $\forall o_1, o_2 \in ext(owl:Thing),\ o_2 \in o_1.owl:sameAs \rightarrow o_1 \in ext(class(o_2)) \wedge o_2 \in ext(class(o_1)) \wedge copy\text{-}values(o_1, o_2)$
r19: $\forall o_1, o_2 \in Obj,\ o_2 \in o_1.owl:sameAs \wedge o_2 \in o_1.owl:differentFrom \rightarrow \square$
r20: $(\forall c \in Cl)\ (\forall r \in c.necessary)\ (\forall b \in r.owl:someValuesFrom)\ (\forall p \in r.owl:onProperty)\ (\forall o \in ext(c))\ \nexists v \in o.p,\ class(v) \sqsubseteq b \rightarrow \exists sk \in ext(b),\ sk \in o.p$

Fig. 1. An OO rule-like syntax of the rules presented in the paper

two multislots *friendOf* and *age* are inserted into the *Person* class (3rd and 6th triple). Notice that although the class is already materialized, the system is able to redefine it with the new slots, presenting a fully dynamic behavior.

• ***r3***, ***r4***: *Define the allowed values for each property*. Each axiom $\top \sqsubseteq \forall P.C$ (the range of P is the class C) is mapped according to the type of P. If P is an object property (*r3*) then it is mapped given the COOL type restriction *INSTANCE-NAME*, taking care of the range class according to [24]. If P is a datatype property (*r4*) then it is mapped given the appropriate COOL datatype restriction *INTEGER*, *SYMBOL*, etc. The *mapDT* function performs OWL to CLIPS datatype conversion. By this rule, the 4th and 7th triples are mapped into the *type* constraint by redefining the properties, i.e. the *age* property has the INTEGER and the *friendOf* property the INSTANCE-NAME type.

• ***r5***: *Implements hierarchical relationships*. Each TBOX assertion of the form $C \sqsubseteq D$ is mapped into a subclass definition using the *is-a* constraint of the COOL *defclass* construct. Both classes should be materialized ($C, D \in Cl$). If there was a subclass relationship in the example ontology, e.g. *Person* $\sqsubseteq$ *Human*, then the *Person* class would be redefined in order to alter the *is-a* constraint into *is-a Human*.

• ***r6***, ***r7***: *Generate the objects of the OO model*. Each axiom $i{:}C$ (i is an instance of C) is mapped into a COOL object of class C. Rule *r6* transforms each triple <s `rdf:type` o> into an object s of the class o, only if o is a materialized class ($o \in Cl$) and there is not any other materialized object with the name s ($s \notin Obj$). By rule *r6*, triples 8 and 9 are mapped into actual objects in the KB. However, OWL allows an object to have multiple class declarations in contrast to the OO modeling principles. This case is treated by rule *r7* which applies only if an object with the same ID already exists in the KB ($s \in Obj$). The object should belong to the intersection of the classes *t.o* and *class*(*t.s*). More specifically, let C and D be two classes and let there be an already implemented ABOX assertion $C(a)$. If a new ABOX assertion $D(a)$ appears, then there are three cases concerning class intersection:

1. If $C \sqsubseteq D$, then the system does not perform any action since the $D(a)$ assertion is satisfiable due to the CLIPS inheritance mechanism.
2. If $D \sqsubseteq C$, then the system redefines the object in order to belong to class D only. Therefore, $D(a)$ and $C(a)$ are satisfiable due to CLIPS inheritance.
3. If neither of the above is true, then the system generates a system class T, where $T \sqsubseteq C$ and $T \sqsubseteq D$ (allowable by CLIPS multiple inheritance mechanism) and implements the ABOX assertion $T(a)$ through which both $C(a)$ and $D(a)$ are satisfiable.

The above algorithm is used in our system whenever an object should belong to more than one classes simultaneously. In the example, if there was an extra triple denoting that `<nick type Human>`, then one of the first two cases of the above algorithm would hold, according to the order the objects would be created: if *nick* was firstly implemented as a *Person* object, then the above triple would be ignored since *Person* $\sqsubseteq$ *Human*. Otherwise, *nick* would be redefined as a *Person* object.

• ***r8***, ***r9***: *Insert values into object properties*. Each $\langle i_1, i_2 \rangle{:}P$ axiom is mapped by inserting the value i_2 into the slot P of the object i_1. The *r8* rule handles the simple case of inserting a value o, which should be a materialized object ($o \in Obj$), into the object property p ($p \in ext(owl:ObjectProperty)$) of the materialized object s ($s \in Obj$). Notice that the slot p should exist in the definition of class of the ob-

1:<Person type Class>	(defclass Person
2:<friendOf type ObjectProperty>	(is-a Thing)
3:<friendOf domain Person>	(multislot friendOf
4:<friendOf range Person>	(type INSTANCE-NAME))
5:<age type DatatypeProperty>	(multislot age (type INTEGER)))
6:<age domain Person>	
7:<age range int>	(make-instance [paul] of Person)
8:<paul type Person>	(make-instance [nick] of Person)
9:<nick type Person>	(send [paul] put-friendOf [nick])
10:<paul friendOf nick>	

Fig. 2. Transformation example.

ject s ($p \in slots(class(s))$). Otherwise, rule *r9* is applied, which mandates the object to also belong to the domain class of the property (in order to inherit it), following the algorithm we have described for multiple class definition objects. By rule *r8*, the 10th triple is mapped as an object value into the *friendOf* slot of the *paul* object. Rules similar to *r8* and *r9* exist for datatype properties as well.

4.1.3 OWL Entailments

We define also transformation rules that implement OWL entailments. To exemplify, we present rule *r10*.

• ***r10***: *Implementation of the rdfs3 entailment*. The rule ensures that the type of the object part *o* of a triple is consistent with the range constraint of the predicate *p* of a triple by forcing the *o* resource to belong to the range constraint class ($o \in ext(p.rdfs:range)$). The rule ensures that all values have the appropriate type before they are inserted into the slots (via *r8*). To give a feeling of the implementation, we present the static CLIPS rule for the *rdfs3* entailment.

```
(defrule rdfs3
  (triple ?s ?p ?o)
  (test (instanceOf ?p owl:ObjectProperty))
  (test (class-existp (send ?p get-rdfs:range)))
=> (create-object ?o (send ?p get-rdfs:range)))
```

The condition matches triples whose predicate is an object property and whose range class already exists. In that case, *?o* is created as an object in the KB via the *create-object* function, which is a functional equivalent of logical rules *r6* (*?o* does not exist) and *r7* (*?o* exists). Thus, if the *nick* object was not defined in the ontology, the system would created it as a *Person* object, the range of the *friendOf* property. A similar rule for the *rdfs2* entailment is based on the domain constraint of a property. In the case of datatype properties, if the value type is inconsistent to the range restriction of the property, the triple is ignored.

4.2 TBOX Reasoning

TBOX reasoning is performed via static rules that apply OWL semantics on class and property definitions.

4.2.1 Class Intersection

The `owl:intersectionOf` construct is treated by defining multiple concurrent subclass relationships. If there is a class C defined as $C \equiv A_1 \sqcap A_2 \sqcap \ldots A_n$, then we define $C \sqsubseteq A_k$, where $1 \le k \le n$, i.e. each A_k class becomes a direct superclass of class C and every object of class C is simultaneously an object of all A_k classes. Notice that class subsumption relationships among A_k are also considered, as already explained. Furthermore, the system stores this as a sufficient condition for class membership, denoting that common objects of all A_k classes are also objects of class C. This information is used during the classification procedure in ABOX reasoning (section 4.3.4), where the common objects of the A_k classes are classified into class C.

4.2.2 Class Union

The `owl:unionOf` construct is also treated by defining subclass relationships. If there is a class C defined as $C \equiv A_1 \sqcup A_2 \sqcup \ldots A_n$, then we define $A_k \sqsubseteq C$, where $1 \le k \le n$, i.e. each A_k class becomes a direct subclass of class C. In that way, the objects of each A_k class belong to class C as well, i.e. $\forall a \mid a \in A_k : a \in C$. Currently, we do not handle the sufficient relation $\forall a \mid a \in C : a \in A_1 \sqcup a \in A_2 \sqcup \ldots \sqcup a \in A_k$. We are investigating ways of applying disjunctive logic programming over the generated OO schema.

4.2.3 Class Equivalence

Class equivalence is another example of the difference between OWL and OO modeling. Since in OO modeling it is infeasible to define mutual subclass relationships among equivalent classes, as the *rdfp12a* and *rdfp12b* entailments denote, we follow an indirect approach.

Let there be a set of n equivalent classes C_n (n>1). The system selects randomly one of the n classes, e.g. class C_d to become the *delegator* class and defines it as a subclass of the rest of the classes, i.e. $C_d \sqsubseteq C_n$ where $n \neq d$. However, this transformation is not enough by itself to capture the complete semantics of class equivalence. Objects of class C_d are also objects of each of the C_n classes, but not vice versa. For that reason we store a sufficient condition stating that an object of any of the C_n classes is also an object of the C_d class. This condition is used later to generate dynamic rules for instance classification which "push" all objects of the C_n classes to the C_d class. In that way, a query to the C_d class retrieves the objects of all C_n classes since their objects have been classified into the C_d class.

4.2.4 Checking Class Consistency

O-DEVICE checks class consistency based on the `owl:disjointWith` property that denotes which classes cannot have hierarchical relationship with each other.

• ***r11***: *Determines class inconsistencies.* The rule checks the consistency of the class hierarchy by examining the values of the `owl:disjointWith` property of class meta-objects. If two classes *b* and *c* are defined to be disjoint with each other (thus *b* belongs to the disjoint slot of *c* ($b \in c.owl:disjointWith$) and vice versa), then the system does not allow the existence of a subclass relationship between them and interrupts the ontology loading procedure with an appropriate error message.

4.2.5 Schema Related Semantics

Specific rules are responsible for creating a complete and valid OO schema. Notice that these rules do not implement any entailment of Table 1. We have defined them based on the formal specification of OWL.

• ***r12***: *Handles domain/range constraints of inverse properties*. For two inverse properties, the domain restriction of the one should be the range of the other and vice versa. More formally, $\forall P_i, P_k: P_i \equiv P_k^-, \top \sqsubseteq \forall P_i^-.C \equiv \top \sqsubseteq \forall P_k.C$.

• ***r13***, ***r14***: *Handle domain/range constraints of subproperties.* A subproperty inherits the domain and range constraints of its superproperties by inserting the values into the corresponding domain and/or range slots. Thus, a property might result in having more than one domain and/or range constraints. These cases are treated by creating an intersection class which acts as a unique domain or range class, as already explained. For example, if a property P has i domain classes, where i>1, i.e. $\top \sqsubseteq \forall P^-.C_i$, then $\top \sqsubseteq \forall P^-.T$, where $T \sqsubseteq C_i$.

• ***r15, r16***: *Define implicit functional and inverse functional properties*. The rules define as functional (*r16*) (or inverse functional (*r15*)) the properties that are defined as the inverse of inverse functional (or functional) properties. For example, if $P_i \equiv P_k^-$ and $\top \sqsubseteq \leq 1 P_k$, then $\top \sqsubseteq \leq 1 P_i^-$.

4.3 ABOX Reasoning

ABOX reasoning is performed via production rules that are dynamically generated based on templates that are "filled" with actual ontology values. In the following sections we present some example templates. All the template rules can be found in [25].

4.3.1 Dynamic Rules for Property Values

User-defined properties may have special semantics according to the class(es) they belong, e.g. `owl:TransitiveProperty`, or because they are related to other properties, e.g. `owl:inverseOf`. In order to handle these property semantics, rules that range over all such properties are needed. As example we present rule *r17*.

r17: *Handles transitive property values (implementation of the rdfp4 entailment)*. This rule actually implements the transitive relation $P(x,y) \wedge P(y,z) \rightarrow P(x,z)$, where P is a transitive property and $x,y,z \in Obj$.

If the above rule was implemented statically in CLIPS a triple loop (i.e. a double join) would be required in the rule condition, as the simplified rule below illustrates. This would be very slow in large ontologies.

```
(defule transitive-property
  (object (is-a owl:TransitiveProperty)(name ?p)(rdfs:domain ?d))
  (object (is-a ?d) (name ?obj1))
  (object (is-a ?d) (name ?obj2))
  (test (member$ ?obj2 (send ?obj1 (sym-cat get- ?p)))
=> (bind $?val1 (send ?obj1 (sym-cat get- ?p)))
   (bind $?val2 (send ?obj2 (sym-cat get- ?p)))
   (send ?obj1 (sym-cat put- ?p) (union$ $?val1 $?val2)))
```

Instead, the system dynamically generates domain specific rules based on template rules that implement special property semantics. The template rule that handles transitive properties can be shown below.

```
(defrule <rule-name>
  (object (is-a <p-domain>) (name ?obj1)
    (<p> $? ?obj2 &: (transitive ?obj1 ?obj2 <p>) $?))
=> (bind $?v1 (send ?obj1 get-<p>))
   (bind $?v2 (send ?obj2 get-<p>))
   (send ?obj1 put-<p> (union$ $?v1 $?v2)))
```

Expressions in bold denote variables that are substituted at runtime by actual ontology values. More precisely, **<p>** denotes a transitive property and **<p-domain>** denotes its domain class. In that way (a) we generate rules that are as specific as possible to the characteristics of a property, restricting the search space of the rule condition and (b) the resulting rules have as less conditional elements as possible, minimizing the cost from multiple joins. The template rule matches objects (`?obj1`) of the domain class, retrieves the values (`?obj2`) of the transitive slot of `?obj1`, exploiting the *message passing infrastructure* of COOL, and calculates the partial transitive closure between the property values of `?obj1` and `?obj2`, without performing any join among objects.

4.3.2 Individual Equality/Inequality

In an OO environment, there is not a direct way to define that two objects are in fact identical. Every object has its unique ID and the only way to achieve such relationship is to implement an indirect mechanism. Our approach results in forcing all the identical objects to have the same values in their corresponding slots (*rdfp11*). An alternative solution could be that, instead of having all the identical objects materialized in the KB, we could select only one to exist as the representative object and all the transformations can be done over this object only. Thus, every query or value insertion that refer to any of the "ghost" objects will be transformed in order to refer to the representative. Although this approach would be more scalable during the loading of an ontology, it is quite customized and does not fully comply with the OO principles since the notion of "subsumed/hidden" objects does not exist. Furthermore, we would impose an extra overhead at query time, since rules should be transformed appropriately in order to refer to the representative object only.

For individuals explicitly defined as identical via `owl:sameAs`, we utilize a static rule since the property is known in advance.

• ***r18***: *Makes identical objects to have the same properties and values*. The rule finds two objects o_1 and o_2 such that o_2 exists in the `owl:sameAs` slot of o_1 and copies all the values of all the properties of both objects to each other, via the *copy-values* procedure which is defined as:

copy-values(o_1,o_2): $\forall s \in slots(class(o_1)) \rightarrow o_1.s = o_2.s$

Since the objects may belong to different classes and thus, may not have the same properties, the system forces them to belong to the same class, before copying, resulting in two objects whose only difference is their name. The name restriction is overcome at the query level where queries traverse objects based on their values and not on their IDs. The same procedure is followed when entailments derive an `owl:sameAs` property (e.g. *rdfp1*, *rdfp2*).

Individual inequality statements are used in order to check consistency, based on the values of the `owl:differentFrom` property and the objects of the `owl:AllDifferent` class. As an example, we present rule *r19*.

• ***r19***: *Checks inconsistencies based on the owl:differentFrom property values*. The rule examines if there is an object o_1 that has in both `owl:sameAs` and `owl:differentFrom` properties the same value o_2 and reports an inconsistency.

4.3.3 Dynamic Rules for Existential Quantifiers

Existential quantifiers are treated by generating Skolem objects based on the properties `owl:someValuesFrom`, `owl:minCardinality`, `owl:cardinality` and `owl:hasValue`. As an example, we present *r20*.

• ***r20***: *Generates existential quantifiers based on the owl:someValuesFrom restriction*. This rule is the logic-like equivalent of the static rule that would generate a Skolem object *sk* for each object *o* of a class *c* with an `owl:someValuesFrom` restriction *r* on property *p*. Notice that classes keep links with their associated restrictions via the *necessary* slot, which stores IDs of instances of the

`owl:Restriction` meta-class. Although restrictions are typically classes in OWL, in O-DEVICE they are not materialized with *defclass* constructs, but their meta-objects are just used for storing their properties. The rest of the restrictions are handled similarly. The actual template rule for rule *r20* is depicted below.

```
(defrule <rule-name>
  (object (is-a <p-domain>)(name ?n)
    (source ?sk &~ SKOLEM)
    (<p> $?val &: (not (exists $?val <c-name>))))
=> (make-skolem-object ?n <p> <c-name>))
```

The rule finds an object `?n` of the restricted class **<p-domain>** and checks if it does not have any values of the appropriate type **<c-name>** for the restricted slot **<p>**. In the action, the rule generates a Skolem object of type **<c-name>** and inserts its ID in the slot **<p>** of the object `?n`.

Notice that Skolem objects need special treatment to avoid non-termination of the derivation procedure. For example, consider the ontology *Person* $\sqsubseteq \exists$*hasParent.Person*. For every *Person* that does not have in its *hasParent* slot a *Person* value, the system will generate a Skolem object of such type and will insert it in the slot. However, for every generated *Person* Skolem object, a new *Person* Skolem object should be generated and the system would go into an endless loop. To solve this, we do not generate Skolem objects for Skolem objects (*source* slot). Although this approach seems simplistic, we argue that we can efficiently handle many real world cases, keeping the complexity of the system low.

4.3.4 Dynamic Rules for Instance Classification

Classification is performed over objects that satisfy the sufficient conditions of the ontology. Appropriate rules change the type of existing objects either by pushing them from a class that is higher in the hierarchy to a more specific class or by pushing them to system generated subclasses which are intersections of existing classes, not hierarchically related. The following template rule generates classification rules.

```
(defrule <rule-name>
  (object (is-a <classes> &~ <class>)
    (name ?n &: (OSR ?n <restrictions>)))
=> (change-object <class> ?name))
```

The expression **<classes>** denotes the classes that an already existing object belongs to and **<class>** is the new class where the object should be classified. The rule finds an object `?n` that belongs simultaneously in all **<classes>** and it is not already an object of **<class>**, since in that case there is no need to activate the rule, and checks if the object satisfies every sufficient condition (**<restrictions>**) of the new class (*OSR* function). In that case, the rule action changes the type of the object using the *change-object* function that implements the algorithm we have already described for multiple type objects in section 4.1.2.

To exemplify, consider the class intersection example of section 4.2.1 defined as $C \equiv A_1 \sqcap A_2 \sqcap \ldots A_n$. In this case, the **<classes>** expression refers to all A_n classes and **<class>** refers to class C where the object should be classified. Since there are not any restrictions to be satisfied, the **<restrictions>** term is *nil*. Thus, an object that satisfies the condition of the rule is classified in class C. The generated classification rule is depicted below.

```
(defrule genA
  (object (is-a A1&A2&…&An &~ C) (name ?n &: (OSR ?n nil)))
=> (change-object C ?name))
```

5 Loading Ontologies

Since O-DEVICE is a memory-based reasoning system, memory size is crucial and imposes a physical threshold on the amount of data that can handle. However, system's performance can be heavily affected by the way the available memory is used.

We have used four ontologies in our experiments. The SEMINTEC [26] ontology is about the financial domain and uses extensively class disjointness, functionality assertions, inverse properties and universal quantifiers. We used a dataset of ~65,000 triples.

VICODI [27] is a project that provides an ontology of European history. The TBOX consists of only subclass relationships, one symmetric property and several subproperties. We used a dataset of ~265,000 triples.

The Lehigh University Benchmark (LUBM) [28] defines an ontology for the university domain. The ontology uses existential quantifiers, intersection of classes and special properties. LUBM provides a tool for generating synthetic OWL data over the ontology. We generated data for one university (LUBM1) with ~105,000 triples.

The University Ontology Benchmark (UOBM) [29] defines two university ontologies for inferencing on OWL Lite and OWL DL by extending the LUBM. In our experiments we use the OWL Lite version that covers sufficiently enough constructs and the UOBM Lite-1 dataset that contains ~245,000 triples. LUBM and UOBM are well known benchmarks, having been used extensively for comparing reasoning engines. A more detailed presentation of benchmarking frameworks can be found in [30].

5.1 Analyzing the Loading Procedure

In an initial implementation, we call *Direct Loading of Triples and Rules* (DLTR), the ontology was transformed into triples that were loaded into CLIPS and rules operated over them in order to create the OO schema and to apply the OWL inference procedure. The drawback of this approach is that rule condition matching (through RETE) involved all the rules simultaneously over the "complete" ontology information. The loading time L_{DLTR} of an ontology Ω of N triples is given by:

$$L_{DLTR} = T_\Omega + L_N + T_N + L_{D,N}, \quad (1)$$

where T_Ω is the transformation time of the ontology into triples, using the ARP Parser [31], L_N is the loading time of N triples into CLIPS, T_N is the transformation time of N triples into an OO schema and $L_{D,N}$ is the loading time of the set D of all the dynamic rules, i.e. the inference rules. T_Ω and L_N are always the same for a specific ontology Ω and are independent of the approach we use for the transformation and inference procedure. Thus, (1) becomes:

$$L_{DLTR} = H + T_N + L_{D,N},$$

where $H = T_\Omega + L_N$.

To enhance system performance, we introduced an *In-*

cremental Loading of Rules (ILR) methodology (Fig. 3). We separate the dynamic rules into ten subsets and each subset is loaded separately. The sets are: transitive (D_1), symmetric (D_2), subproperty (D_3), inverse (D_4), equivalent (D_5), functional (and min cardinality) (D_6), inverse functional (D_7), universal quantifiers (D_8), skolem/existential (D_9) and classification (D_{10}). We have excluded from the analysis the individual equality rule (`owl:sameAs` property), since the rule is static and ontology-independent. Rule subsets are applied in a circular mode, to cope with the missing rule activations due to incremental rule loading, until no activation is detected. The loading time L_{ILR} of an ontology Ω of N triples is given by the following formula, where a is the number of cycles of the inference procedure and $L_{D_n,N}$ is the loading time of the n-th set of dynamic rules over the generated OO schema from the transformation of N triples: $D = \cup D_i$, where D is the complete set of all the dynamic rules.

$$L_{ILR} = H + T_N + a\sum_{i=1}^{n} L_{D_i,N},$$

Experiments have shown that the incremental loading of dynamic rules performs better, even if the ABOX reasoning procedure needs to be applied many times. When the complete set of rules is loaded, the firing of one of them causes the pattern matching procedure to be executed over all rules in order to determine rule activations/deactivations. By loading each time a portion of the rule set, the pattern matching procedure operates faster, even if the system spends extra time in order to apply the inference rules in a circular mode. In other words:

$$a(L_{D1,N} + L_{D2,N} + ... + L_{Dn,N}) < L_{D,N}.$$

However, even if rules were applied incrementally, each time rule conditions had to be matched over a large number of objects leading to a poor performance. To cope with this problem, we have also implemented an *Incremental Loading of Triples* (ILT) methodology (Fig. 4). The system incrementally loads sets of q triples, where q is a predefined value, and then applies the ILR methodology over the currently loaded data in order to create each time a portion of the OO schema. Thus, the overall reasoning procedure consists of N/q cycles. The loading time is:

$$L_{ILT+ILR} = T_\Omega + (N/q)L_q + (N/q)T_q + a(N/q)\sum_{i=1}^{n} L_{D_i,q}, \qquad (2)$$

where $(N/q)L_q$ is the sum of the loading times of each q triples incrementally into CLIPS, $(N/q)T_q$ is the sum of the transformation times of q triples incrementally into the OO schema and the last factor is the sum of the loading times of the dynamic rules. Assuming that bulk loading N triples into memory is the same as loading them incrementally, i.e. $(N/q)L_q = L_N$, (2) becomes:

$$L_{ILT+ILR} = H + (N/q)[T_q + a\sum_{i=1}^{n} L_{D_i,q}].$$

L refers only to loading triples without applying any rule.

Experiments have shown that for an appropriate q value (a) the incremental loading of dynamic rules over a portion of the OO schema is faster than the incremental loading over the whole set, since rules are applied to a smaller set of objects, and (b) the incremental transformation procedure of triples into an OO schema is faster than transforming all the triples at once, since the transformation rules are applied in a portion of the triples and, thus, activated faster. In other words:

$$a(N/q)\sum_{i=1}^{n} L_{D_i,q} < a\sum_{i=1}^{n} L_{D_i,N}\ ,\ (N/q)T_q < T_N.$$

5.1.1 Determining the q Value

In order to discover the factors that influence q, we have conducted experiments with different ontologies. We loaded each ontology and for each run we used a different q value. The results we obtained are depicted in Fig. 5. Table 2 shows the approximate number of generated objects in each cycle according to the number q of triples.

TABLE 2
Approximate Number of Generated Objects in Each Cycle

	5,000	10,000	20,000	30,000
UOBM Lite1	950	1,900	**3,950**	5,900
LUBM1	975	2,050	**4,200**	6,420
SEMINTEC	1,380	2,760	**5,500**	8,450
VICODI	**1,550**	3,150	6,350	9,800

We observed that three out of the four ontologies were loaded faster with a q value approximately equal to 20,000, where the number of generated objects in each cycle varies from 4,000 to 5,500 objects. The VICODI ontology exhibits a different behavior, achieving the best loading time for q=5,000. Furthermore, increasing q affects heavily the loading time of VICODI in contrast to the other ontologies where the loading time increase is smoother. In order to explain this difference, we examined the dynamic rules that were generated.

We conclude that the optimal q value is affected by the percentage of the overall schema the rules have to traverse, i.e. the *is-a* part of a rule that denotes the class of the matched objects. Furthermore, the amount of overhead is relevant to the type of the dynamic rule: a transitive rule imposes greater overhead than a symmetric one.

For the VICODI ontology, the rules that are generated traverse many objects since they match objects of classes

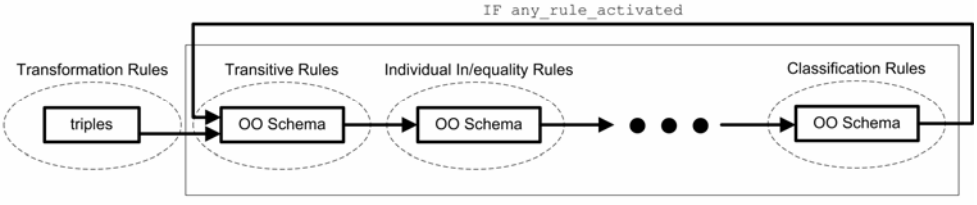

Fig. 3. The Incremental Loading of Rules architecture.

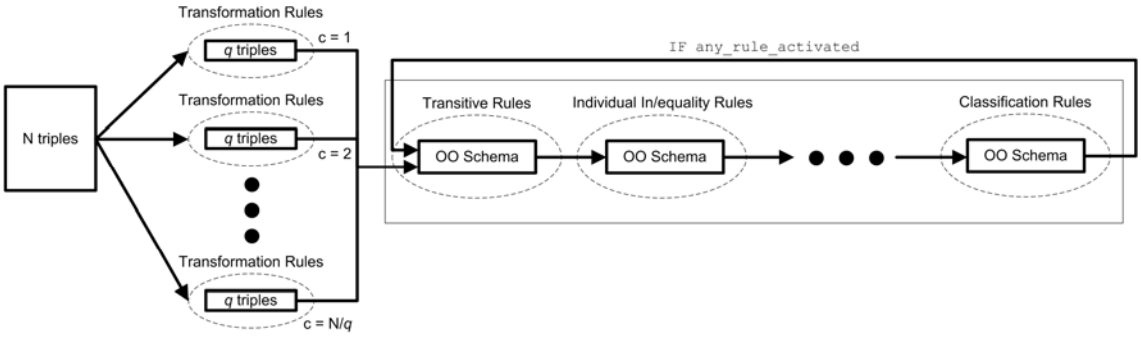

Fig. 4. The combination of the ILT and ILR architectures.

high in the hierarchy and for that reason the best loading time is observed by inserting only 1,500 objects. However, the inference procedure for UOBM Lite-1 and LUBM1 datasets is not heavily affected by the increase of q. We observe smooth changes of loading times as q increases. In fact, the loading performance decreases with smaller q values, since the time the system gains by applying the rules in a smaller number of objects is overbalanced by the time it spends to apply the dynamic rules in more cycles. The same holds for the SEMINTEC ontology, where the *is-a* part of the rules refer to a small portion of the objects and there is no need for a small q value.

We conclude that the system can efficiently handle 4,000 to 6,000 newly generated objects in each cycle, and by default, the system operates by loading 20,000 triples in each cycle. However, in order to enhance system's performance in cases such as the VICODI ontology, we present a heuristic approach for automatically adjusting the q value. We define a metric p that represents the degree of ontology complexity in terms of the overhead that the corresponding dynamic rules impose in the system's ABOX reasoning procedure. It is worth mentioning, that the q value that we estimate is not an optimal one. The reason is that it cannot represent the actual number of instances that would be loaded. For example, if there are many properties in the ontology, many of the triples would refer to property values of the objects and not to actual definitions of objects. Thus, for different ontologies, the same q value would result in the generation of different number of objects in each cycle.

5.1.2 Estimating the p Metric

We define the p metric for a rule as:

$$p_n = w_n k_n , \tag{3}$$

where p_n is the p metric for a rule n, w_n is a weight that characterizes the class where a rule n refers to in its *is-a* part and k_n is a weight that represents how much the rule n is influenced by the weight w_n.

To calculate the w weight for a class i, we have developed a module that reads the class hierarchy and assigns a weight in every class, depending on its position in the class hierarchy and the total number of classes. The assignment of such a weight is based on the heuristic observation that a class is likely to contain a large number of objects if it has many subclasses. Furthermore, we made the assumption that classes have a uniform number of objects. That is $w_i = s_i/tc$, where w_i is the weight for a class i, s_i is the number of subclasses of i and tc is the total number of user-defined classes. In that way, each weight w is a number between 0 and 1 and denotes an estimation for a class to have a large number of objects. For example, the `owl:Thing` class will be assigned with w=1 since all objects belong to this class. Leaf classes are assigned w=0.

The k_n weight denotes how much a rule n is affected by the w_n weight. Each dynamic rule is characterized by different complexity, e.g. the calculation of the transitive closure is a time consuming process and it is affected heavily by the portion of the schema it is computed on. On the other hand, the Skolemization process cannot be considered as a complex one. We have defined weights for each dynamic rule type, depicted in Table 3.

TABLE 3
The Assignment of k Weights in Different Types of Rules

	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}
k_n	0.9	0.7	0.4	0.4	0.4	0.6	1.0	0.2	0.2	0.2

We consider inverse functional and transitive rules as the most complex, especially the first ones since they impose a join of objects. However, for the universal quantifier, Skolemization and classification rules we have assigned a small weight, since the implemented semantics are handled fast by the rule engine.

We generalize (3) to a subset of the dynamic rules of the same type as:

$$p_{D_n} = \max\{w_n\} k_n ,$$

where p_{D_n} is the p metric for a subset D_n, $\max\{w_n\}$ is the maximum class weight w present in the whole D_n subset and k_n is the k weight for the rules in the subset D_n. There-

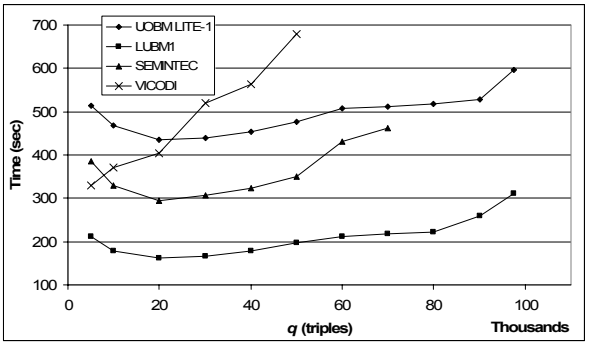

Fig. 5. Loading times for different q values.

fore, the p metric of the D_n rule subset depends only on the maximum w class weight referred by the corresponding rules. Thus, p is given by equation (4) as the maximum p_{D_n} of all rule subsets D_n.

$$p = \max\{\max\{w_n\}k_n\}. \quad (4)$$

5.1.3 The Relation of q to the p Metric

Equation (4) imposes that p is a value between 0 and 1 in the worst case. The value 0 denotes that the ontology does not require any special ABOX reasoning ability, e.g. an ontology that defines only subclass relationships and simple object or datatype properties. In that case, p = 0 since $\max\{w_n\}k_n$ = 0 for $1 \le n \le 10$. The worst case occurs when an ontology requires to traverse all the schema objects in order to apply an inverse functional property, i.e. $\max\{w_7\}k_7 = 1$. We set a threshold to p value equal to 0.6. This is an arbitrary choice, based on the experimental results. If the p value for an ontology is smaller than this threshold, then the loading is performed with the default value q=20,000. Otherwise, the system sets q=5,000. By applying this methodology over the four tested ontologies, we obtain the p values of Table 4. The calculation of the p metric does not impose an extra overhead to the loading procedure, since it takes only a few milliseconds.

TABLE 4
The p Metric for the Tested Ontologies

	UOBM	LUBM	SEMINTEC	VICODI
p	0.48	0.19	0.1	0.7

The method for calculating p is based on heuristic assumptions and cannot be used as an absolute criterion for the actual complexity of an ontology. Its usefulness is restricted only to detect extreme cases where the inference requirements of an ontology result in demanding rules, as far as objects traverse requirements are concerned. For example, in flat ontologies, many classes will be assigned with w=0, since they will not have subclasses and s=0. But usually real world ontologies have a sufficient hierarchy in order to find a good estimation for p.

5.2 Ontology Loading Results

We apply our transformation methodologies, namely DLTR, ILR and ILT+ILR in 4 datasets. The loading times depicted in Fig. 6 incorporate the mean time after 5 runs needed to transform the ontology into triples, to load the triples into the system and to perform the transformation/inference procedure. The ILT+ILR approach performs better than the other two in all datasets. It is worth

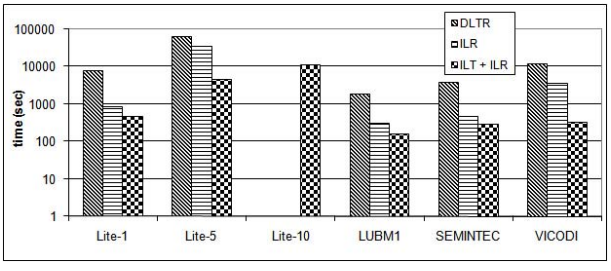

Fig. 6. The loading results of the three loading approaches.

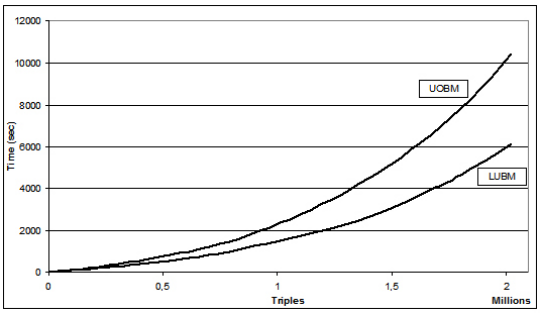

Fig. 7. Loading time according to the number of triples.

mentioning that the ILT+ILR methodology was the only one that managed to load the UOBM Lite-10 dataset that contains ~2 million triples in a reasonable time limit.

In order to examine the complexity of the loading procedure, we generated a dataset for LUBM with almost the same triples as the UOBM Lite-10 dataset. Fig. 7 depicts how the loading times of the two datasets are affected by the number of triples. As we have already described (section 4.3), O-DEVICE generates dynamically ABOX inference production rules according to the defined constructs in an ontology. Since each entailment has its own complexity, the overall complexity depends on the generated production rules, i.e. depends on the TBOX. LUBM requires less inferencing capabilities than UOBM and the inferencing procedure terminates faster. Notice that an absolute linear behavior cannot be achieved since before the creation of an object or the insertion of a property value into an object's slot, the system should check if the object exists. As the number of objects increases, this check becomes more time consuming. However, the implementation is totally based on the native CLIPS mechanisms without having implemented any special structures for handling OWL constructs or to treat the large number of objects. We believe that more sophisticated indexing methods would increase the performance considerably.

6 Query Language

The deductive rule language of O-DEVICE supports querying over OWL instances represented as objects. The conclusions of deductive rules represent derived classes, whose objects are generated by evaluating these rules over the current set of objects. Each deductive rule is implemented as a CLIPS production rule that inserts a derived object when the condition of the deductive rule is satisfied. The query language aims at simplifying the definition of queries, following a simpler syntax than CLIPS or to eliminate any lisp-like syntax through the RuleML-like module that supports [24].

As an example of the deductive rule language, we give the first query of UOBM which retrieves all the instances of the class `UndergraduateStudent` that have the value `Course0` in the property `takesCourse`.

```
1:(deductiverule r1
2: ?id <- (UndergraduateStudent (takesCourse $? [Course0] $?))
3:=> (result (uGradStud ?id)))
```

Line 2 defines the condition of the rule. Variable `?id`

refers to the ID of the object that satisfies the restriction on the property `takesCourse`, namely to have `[Course0]` among its values. Line 3 defines the conclusion of the rule, namely the pattern of the derived object, having on the slot `uGradStud` the name of the object that satisfies the condition. Thus, the result of this query consists of objects of the class `result` that have on the property `uGradStud` the names of the actual objects that satisfy the condition of the rule. The deductive rule is transformed into the following production rule.

```
(defrule gen234
  (object (is-a UndergraduateStudent) (name ?id)
    (takesCourse $? ?gen5 &: (contains ?gen5 [Course0]) $?))
  (test (not (instance-existp (sym-cat result ?id))))
=> (bind ?oid (sym-cat result ?id))
   (make-instance ?oid of result (uGradStud ?id)))
```

The rule, instead of matching directly `UndergraduateStudent` objects that have the value `[Course0]` in the `takesCourse` slot, uses an intermediate variable `?gen5` and the function *contains* that first checks if the object matched by the variable matches the ID of the actual object variable. If not, it checks if the object matched by the variable has an `owl:sameAs` relationship with the actual object variable, i.e. if the object `Course0` exists in the `owl:sameAs` values of the matched object by the variable (or vice-versa). If this check fails too, so does the function. More formally, the *contains* function returns true if:

$$?av = ?qv \vee ?qv \in ?av.owl{:}sameAs.$$

In that way, the rule also matches object values based on the `owl:sameAs` information, overcoming the problem of OO programming languages, where objects with different IDs are necessarily different objects. Notice that identical objects are retrieved using the *message passing mechanism* of CLIPS avoiding thus joins.

7 Experimental Results

We tested O-DEVICE query performance using LUBM1 and UOBM Lite-1 datasets that define 14 and 13 extensional queries, respectively. We also run the same experiments on OWLIM (v2.8.4) and OWLJessKB rule-based reasoners, Pellet (v1.3) DL reasoner and KAON2 (build-2007-01-06) datalog-driven reasoner. We have measured the time needed to load the ontologies (time spent in any processing of the ontology, such as parsing and reasoning) and to answer the queries (time spent to load, process and answer queries). We also measured the maximum memory consumption of each system to complete the benchmarks. The experiments run on a laptop with AMD Turion ML-34, 1.8 GHz processor, 1 GB RAM and JAVA EE 5 SDK, setting maximum heap size 700 MB.

Fig. 8 depicts the loading times. All systems, except KAON2 and Pellet, follow a complete materialization approach, i.e. they apply rules at loading time in order to derive the implicit information. KAON2 and Pellet handle semantic information on demand. Thus, they load faster the ontologies than OWLIM, O-DEVICE and OWLJessKB. However, OWLIM loads ontologies considerably fast. We believe that this happens due to the sophisticated

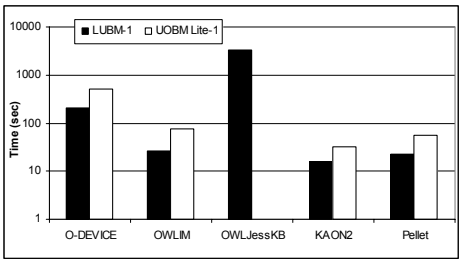

Fig. 8. Loading times.

indexing methods and the dedicated rule OWL reasoning engine that OWLIM uses (TRREE [32]). O-DEVICE and OWLJessKB use native rule engines (CLIPS and Jess respectively) that are not OWL-oriented. But O-DEVICE loads ontologies considerably faster than OWLJessKB which could not load UOBM Lite-1 due to memory limitation. The transformation of ontologies into an OO schema and the utilization of dynamically generated rules perform better than applying rules directly over triples.

Fig. 9 depicts the maximum memory requirements of each system for both loading and querying. Notice that the measurement of KAON2 for the UOBM Lite-1 ontology incorporates only the loading memory requirements, since it did not manage to answer any query due to memory limitation. O-DEVICE requires the least memory, whereas OWLJessKB requires almost 680 MB.

Fig. 10 and Fig. 11 depict the query response times for LUBM1 and UOBM Lite-1, respectively. All systems returned the same sets of instances, except OWLJessKB which, in some queries, returned some incorrect instances. Concerning the LUBM1 ontology, OWLIM performs the best, whereas KAON2 and Pellet need the most time to answer queries, since they perform OWL inferencing at query time. O-DEVICE and OWLJessKB exhibit similar performance; some queries run faster in one system and some on the other, with O-DEVICE spending much time in answering query 9. We believe that the reason of this behavior lies on technical aspects of the implementation of both systems. Concerning the UOBM Lite-1 ontology, KAON2 threw a heap error exception in all queries. Furthermore, OWLIM failed to answer query 9 (we have used the default settings of OWLIM distribution for the UOBM ontology), returning no results, whereas O-DEVICE managed to answer it within 156 seconds. Moreover, Pellet failed to answer queries 4, 9, 11, 12 and 13, throwing a nominal exception.

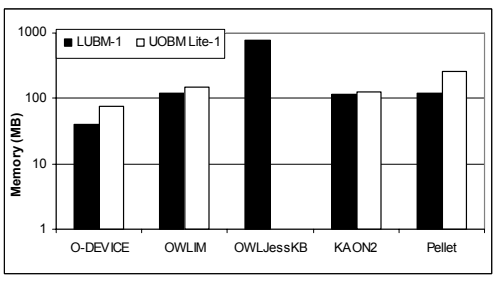

Fig. 9. Memory requirements.

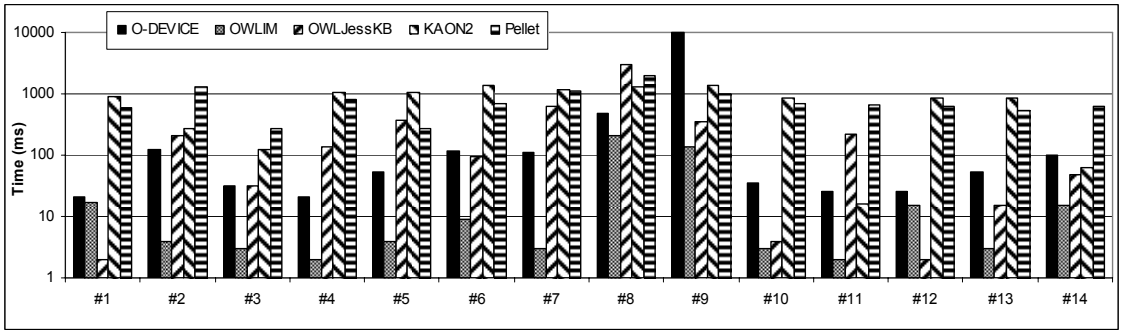

Fig. 10. Query response times for the LUBM1 dataset.

In order to give a gist of how O-DEVICE performs in different types of queries, we analyze the query results of the LUBM1 dataset. The complexity for a query in O-DEVICE depends on two factors: (a) the number of joined objects in the query condition and (b) the number of objects that the query needs to traverse, i.e. the class(es) the query refers to. We can see that query response times of O-DEVICE are clustered into three groups. Queries of the first group (1, 3, 4, 5, 10, 11, 12) are executed faster since they do not impose any join of objects, apart from query 12 which although requires a join, refers to classes with small number of instances. A typical example is query 5:

```
(deductiverule lubm-q5
  ?o1 <- (Person (memberOf
                  $? [http://www.Department0.University0.edu] $?))
=> (result (name ?o1)))
```

Queries of the second group (2, 6, 7, 8, 13, 14) require more time to be answered. Queries 2, 13 and 14, although they do not impose any join, they refer to classes with large number of objects, e.g. *Person*, in contrast to queries of the first group. Queries 2, 7 and 8 require a join and in addition to the large number of objects of the referred class, increase the response time compared to the query 12. A typical example of this cluster is query 8:

```
(deductiverule lubm-q8
  ?o2 <- (Department (subOrganizationOf
                      $? [http://www.University0.edu] $?))
  ?o1 <- (Student (memberOf $? ?o2 $?) (emailAddress ?mail))
=> (result (x ?o1) (y ?o2) (z ?mail)))
```

Finally, query 9 is the most complicated query, since it joins instances from three classes, one of which contains the larger number of instances (*Student* class). The R-DEVICE syntax of all LUBM queries is available in [25].

8 Related Work

Pellet [3], RacerPro [4] and Fact++ [33] are well known DL reasoners. Pellet is based on the tableaux algorithms developed for expressive Description Logics and it is complete on SHIN(D) and SHON(D). It supports an ABox query answering module using the "rolling-up" technique. Racer implements a highly optimized tableaux calculus for a very expressive description logic. It also implements a description logic query language for ABox individuals, named nRQL [34]. FaCT++ uses a variation of the established FaCT tableaux algorithms for description logic inferencing. FaCT and FaCT++ do not directly support ABOX reasoning. We have chosen Pellet as an optimized DL reasoner for ABOX queries.

Vampire [13] is a FOL engine, exploiting several techniques to improve its performance such as optimized algorithms for backward and forward subsumption, indexing and discrimination trees. In [35] a comparison between Vampire and the Fact++ reasoner is presented.

KAON2 [6] is a DL reasoner where reasoning is implemented by novel algorithms which reduce a SHIQ(D) KB to a disjunctive datalog program [7]. It is argued that this approach is fast for large ABoxes, due to certain optimization techniques, such as magic sets or join-order optimizations. The experiments have shown that KAON2 requires a considerable amount of memory.

OWLJessKB [9], a successor of DAMLJessKB [36], is a memory-based reasoner for OWL, implemented using the

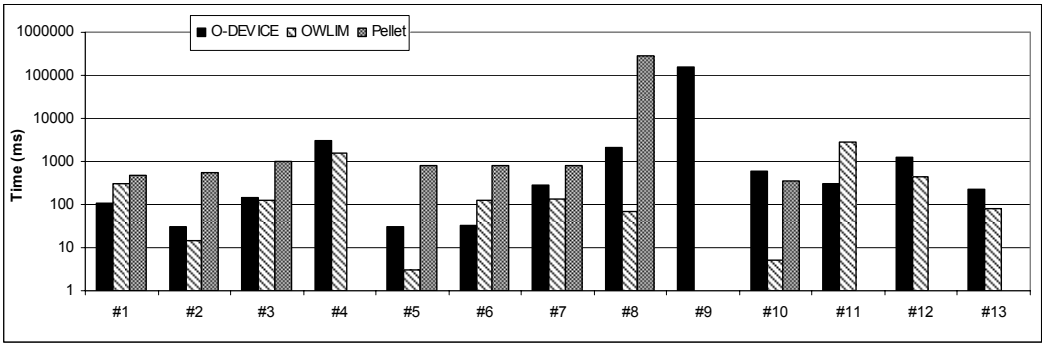

Fig. 11. Query response times for the UOBM Lite1 dataset.

Jess [37] rule engine. It translates RDF triples directly into facts and uses production rules to implements OWL entailments. We have shown in the experiments that such a trivial approach is functional for relatively small ontologies but suffers from scalability and memory limitations.

Bossam [38] is a RETE-based forward-chaining production rule engine that has an RDF logic-like rule language, called Buchingae. We were unable to load the LUBM1 ontology due to memory limitations.

Corese [14] is an engine that internally works on conceptual graphs. When matching a query with an annotation, both RDF graphs and their schema are translated in the conceptual graph model. Through this translation, Corese takes advantage of previous work of the KR community leading to reasoning capabilities of this language.

To the best of our knowledge, OWLIM [8] is the fastest memory-based system that performs reasoning based on forward chaining of entailment rules. OWLIM uses the dedicated OWL TRREE engine and the most expressive language supported is a combination of limited OWL Lite and unconstrained RDFS.

F-OWL [11], Ontobroker [12] and Florid [39] are systems that employee F-Logic [10] for data definition and querying. The main difference with O-DEVICE is that they use a frame-based language only for representing the information whereas we build a native OO schema that complies with OO principles. For example, F-OWL uses the Flora2 [40] system that translates a dialect of F-Logic into the XSB [41] deductive engine, i.e. into Tabled Prolog. To understand the difference, consider for example instance equality. In Flora2 (and in F-OWL), instance equality can be modelled by substituting a term with an equivalent one based on the facts. In O-DEVICE we create multiple objects with different IDs and we treat them as same through the query language, making our approach general and applicable in any OO environment.

One more example is class equivalence, which in F-OWL is defined as:

A[owl_*equivalentClass*->>*B*] :-
A[*rdfs_subClassOf*->>*B*], *B*[*rdfs_subClassOf*->>*A*].

This is feasible, since subclass relationships are modelled as facts. However, in an OO environment, such mutual subclass relationships cannot be modelled because subclass graph cycles are forbidden.

9 Conclusions and Future Work

In this paper we have described a memory-based rule system for inferencing about and querying ontologies expressed in a more expressive set than OWL Lite. We build an OO model of the ontologies into the COOL language of CLIPS and production rules match objects instead of triples. In that way, we enable a well-known and highly reliable rule engine to handle OWL semantics.

In our approach we do not use static production rules for every OWL construct but we follow a *Dynamic Rule Generation* approach where domain specific rules are generated according to ontology characteristics. The rules are simpler than the corresponding static rules because they contain fewer condition elements and thus they are activated faster. Furthermore, we have adopted an incremental triple loading and rule application procedure in order to avoid the excessive memory management system activities that lead to poorer performance.

A memory-based reasoning system should exploit the available memory in an efficient way in order to load the larger possible set of ontologies. Experimental results have shown that O-DEVICE requires less memory than the rest of tested systems, being able to load ontologies and answer queries that other system failed due to memory limitations. Although OWLIM is the fastest rule-based reasoner, we argue that the reusability of an *existing rule engine* gives great potentials, based on the practicality, efficiency and optimized techniques that has obtained thought the years of the development.

For the future, we plan to introduce more sophisticated indexing methods in order to improve the retrieval performance, without sacrificing the practical aspect of the approach, since currently we are based only on native CLIPS mechanisms. One such solution would be the substitution of the hash function that CLIPS uses for determining the existence of an object, since it might be inappropriate to handle so large number of objects or two implement dedicated structures in order, for example, to further index identical individual. In that way, we could improve both loading and querying performance. Furthermore, we plan to connect O-DEVICE with [42], a visual tool for defining queries using a RuleML-like syntax. Finally, we plan to use the reasoner as the core system of a Web service discovery and composition framework [43] [44] based on OWL-S [45] descriptions.

Acknowledgment

This work was partially supported by a PENED program (EPAN M.8.3.1, No. 03EΔ73), jointly funded by the European Union and the Greek Government (General Secretariat of Research and Technology/GSRT) and by a NON-EUROPE project (GSRT - 05 NON EU 423).

References

[1] W3C Semantic Web Activity, http://www.w3.org/2001/sw/
[2] Web Ontology Language, http://www.w3.org/2004/ OWL/
[3] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur and Y. Katz, "Pellet: A Practical OWL DL Reasoner", *Journal of Web Semantics*, 2007.
[4] V. Haarslev, R. Möller, "Racer: A Core Inference Engine for the Semantic Web", *2nd Int. Workshop on Evaluation of Ontology-based Tools*, Florida, USA, pp. 27–36, 2003
[5] F. Baader, U. Sattler, "An Overview of Tableau Algorithms for Description Logics", *Studia Logica*, vol. 69, pp. 5-40, 2001
[6] B. Motik, R. Studer, "KAON2 – A Scalable Reasoning Tool for the Semantic Web", *Proc. 2nd ESWC*, Heraklion, Greece, 2005
[7] U. Hustadt, B. Motik, U. Sattler, "Reducing SHIQ Description Logic to Disjunctive Datalog Programs", *Proc. KR2004*, Whistler, Canada, pp. 152-162, 2004.
[8] A. Kiryakov, D. Ognyanov, D. Manov, "OWLIM - a Pragmatic Semantic Repository for OWL", *Proc. Workshop Scalable Semantic Web Knowledge Base Systems*, USA, 2005
[9] OWLJessKB: A Semantic Web Reasoning Tool, http://edge.cs.drexel.edu/assemblies/software/owljesskb/

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

[11] Y. Zou, T. Finin and H. Chen, "F-OWL: an Inference Engine for Semantic Web", *Proc. 3rd Inter. Workshop Formal Approaches to Agent-Based Systems*, Greenbelt, USA, 2004

[12] S. Decker, M. Erdmann, D. Fensel, R. Studer, "Ontobroker: Ontology based access to distributed and semi-structured information", *Database Semantics-Semantic Issues in Multimedia Systems, IFIP Conf. Proc.,* vol. 138, 1998, pp. 351–369.

[13] D. Tsarkov, A. Riazanov, S. Bechhofer, I. Horrocks, "Using Vampire to reason with OWL", *Proc. ISWC 2004,* pp. 471-485.

[14] O. Corby, R. Dieng-Kuntz, C. Faron-Zucker, "Querying the Semantic Web with the CORESE search engine", *Proc. PAIS'2004*, Valencia, Spain, IOS Press, pp. 705-709.

[15] CLIPS, http://www.ghg.net/clips

[16] G. Meditskos, N. Bassiliades, "Towards an Object-Oriented Reasoning System for OWL", *Proc. Workshop OWL Experiences and Directions*, Galway, Ireland, 2005, CEUR, Vol. 188.

[17] G. Meditskos, N. Bassiliades, "O-DEVICE: An Object-Oriented Knowledge Base System for OWL Ontologies", *Proc. 4th Hellenic Conf. on Artificial Intelligence*, Crete, Greece, 2006.

[18] S. Bechhofer, R. Moller, P. Crowther, "The DIG description interface", *International Workshop on Description Logics*, 2003.

[19] SWRL, http://www.w3.org/Submission/SWRL/.

[20] SPARQL, http://www.w3.org/TR/ rdf-sparql-query/

[21] B. Motik, I. Horrocks, R. Rosati, U. Sattler, "Can OWL and Logic Live Together Happily Ever After?", *Proc. 5th ISWC*, Athens, USA, 2006.

[22] H.J. Horst, "Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary", *Journal of Web Semantics*, vol. 3, pp. 79-115, 2005

[23] OWL Web Ontology Language Reference (Appendix B), http://www.w3.org/TR/owl-ref/

[24] N. Bassiliades, I. Vlahavas, "R-DEVICE: An Object-Oriented Knowledge Base System for RDF Metadata", *Int. Journal on Semantic Web and Information Systems*, 2(2), pp. 24-90, 2006

[25] The O-DEVICE System, http://iskp.csd.auth.gr/systems/o-device/o-device.html

[26] SEMINTEC - Semantically-enabled data mining techniques, http://www.cs.put.poznan.pl/alawrynowicz/semintec.htm

[27] VICODI Ontology, http://www.vicodi.org/

[28] Y. Guo, Z. Pan, J. Heflin, "LUBM: A Benchmark for OWL Knowledge Base Systems", *Journal of Web Semantics*, 3(2), pp. 158-182, 2005

[29] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, S. Liu, "Towards a Complete OWL Ontology Benchmark", *Proc. 3rd ESWC*, Budva, Montenegro, pp. 125–139, 2006

[30] Y. Guo, A. Qasem, Z. Pan, J. Heflin, "A Requirements Driven Framework for Benchmarking Semantic Web Knowledge Base Systems", *IEEE TKDE*, vol. 19, Feb. 2007

[31] B. McBride, "Jena: Implementing the RDF Model and Syntax Specification", *2nd Int. Workshop on the Semantic Web*, 2001

[32] TRREE - Triple Reasoning and Rule Entailment Engine, http://www.ontotext.com/trree/

[33] D. Tsarkov, I. Horrocks, "FaCT++ Description Logic Reasoner: System Description", *Int. Conf. on Automated Reasoning*, 2006

[34] V. Haarslev, R. Möller, M. Wessel, "Querying the Semantic Web with Racer + nRQL", *Proc. 3rd Int. Workshop on Applications of Description Logics*, Ulm, Germany, 2004

[35] D. Tsarkov, I. Horrocks, "DL reasoner vs. first-order prover", *Proc. 2003 DL Workshop*, CEUR, vol. 81, pp. 152–159, 2003.

[36] J.B. Kopena, W.C. Regli, "DAMLJessKB: a tool for reasoning with the Semantic Web", *Proc. 2nd ISWC*, 2003.

[37] Jess, http://herzberg.ca.sandia.gov/jess

[38] J. Minsu, J.C Sohn, "Bossam: An Extended Rule Engine for OWL Inferencing, *Proc. RuleML 2004*, pp. 128-138, Japan, 2004.

[39] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, C. Schlepphorst, "Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective", *Information Systems, 23 (8), Special Issue on Semistructured Data*, pp. 589-612, 1998.

[40] G. Yang, M. Kifer, C. Zhao, "FLORA-2: A rule-based knowledge representation and inference infrastructure for the semantic web", *Proc. 2nd ODBASE*, 2003

[41] K. Sagonas, T. Swift, D.S. Warren, "XSB as an efficient deductive database engine", *Int. Conf. on the Management of Data*, pp. 442–453, ACM Press, 1994

[42] N. Bassiliades, E. Kontopoulos, G. Antoniou, "A Visual Environment for Developing Defeasible Rule Bases for the Semantic Web", *Proc. RuleML-2005*, Galway, Ireland, pp. 172-186.

[43] G. Meditskos, N. Bassiliades, "A Semantic Web Service Discovery and Composition Prototype Framework Using Production Rules", *OWL-S: Experiences and Future Developments workshop of 4th ESWC*, Innsbruck, Austria, June, 2007

[44] G. Meditskos, N. Bassiliades, "Object-Oriented Similarity Measures for Semantic Web Service Matchmaking", *5th IEEE ECOWS, Halle (Saale), Germany, November 26-28, 2007 (to appear).*

[45] OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/

Georgios Meditskos received the BSc degree in Computer Science in 2004 and the MSc degree in Computer Science in 2007 from the Aristotle University of Thessaloniki, Greece. Since 2004, he has been a PhD student in Computer Science at the Aristotle University of Thessaloniki. His research interests include Semantic Web, Semantic Web Services and Artificial Intelligence.

Nick Bassiliades received the PhD degree in parallel knowledge base systems in 1998 from the Department of Informatics, Aristotle University, Thessaloniki, Greece, where he is currently an assistant professor. His research interests include knowledge base systems, rule systems and the semantic web. He has published over 70 papers at journals, conferences and books and co-authored an international book on Parallel, Object-Oriented, and Active Knowledge Base Systems and a Greek book on Artificial Intelligence. He has been involved in projects concerning knowledge based systems, intelligent agents, e-learning, web services, semantic web, etc. He is a member of the Board of the Greek Artificial Intelligence Society and also a member of the Greek Computer Society, the IEEE and the ACM.