Towards an Object-Oriented Reasoning System for OWL

Georgios Meditskos, Nick Bassiliades

Department of Informatics, Aristotle University of Thessaloniki, Greece
{gmeditsk, nbassili}@csd.auth.gr

Abstract. In this paper we present O-DEVICE, a deductive object-oriented knowledge base system for reasoning over OWL documents. O-DEVICE imports OWL documents into the CLIPS production rule system by transforming OWL ontologies into an object-oriented schema of the CLIPS Object-Oriented Language (COOL) and instances of OWL classes into COOL objects. The purpose of this transformation is to be able to use a deductive object-oriented rule language for reasoning about OWL data. The O-DEVICE data model for OWL ontologies maps classes to classes, resources to objects, property types to class slot (or attribute) definitions and encapsulates resource properties inside resource objects, as traditional OO attributes (or slots). In this way, when accessing properties of a single resource, few joins are required. O-DEVICE is an extension of a previous system, called R-DEVICE, which effectively maps RDF Schema and data into COOL objects and then reasons over RDF data using a deductive object-oriented rule language.

1 Introduction

Semantic Web is the next step of evolution for the Web, where information is given well-defined meaning, enabling computers and people to work in better cooperation. Ontologies can be considered as a primary key towards this goal since they provide a controlled vocabulary of concepts, each with an explicitly defined and machine processable semantics.

Furthermore, a lot of effort is undertaken to define a rule language for the Semantic Web on top of ontologies in order to combine already existing information and deduce new knowledge. Currently, RuleML [6] is the main standardization effort for rules on the Web to specify queries and inferences in Web ontologies, mappings between ontologies, and dynamic Web behaviors of workflows, services, and agents.

One approach to implement a rule system on top of the Semantic Web ontology layer is to start from scratch and build inference engines that draw conclusions directly on the OWL data model. However, such an approach tends to throw away decades of research and development on efficient and robust rule engines. In this paper we follow a different approach: we re-use an existing rule system (CLIPS [8]) for reasoning on top of OWL data. However, before an existing rule system is used, careful design must be made on how OWL data and semantics are going to be treated in the host system.

More specifically, we describe O-DEVICE, a system for inferencing over (on top of) OWL documents. The system transforms OWL ontologies into an object-oriented schema of the OO programming language provided within CLIPS, called COOL [8], and OWL data into objects. The O-DEVICE data model maps OWL classes to classes, OWL resources to objects, OWL property types to class slot (or attribute) definitions and encapsulates OWL resource properties inside objects, as traditional OO attributes (or slots). In this way, when accessing properties of a single resource few joins are required. The system also features a powerful deductive rule language which supports inferencing over the transformed OWL descriptions. Users can either use this deductive language to express queries or a RuleML-like syntax.

O-DEVICE is an extension of a previous system, called R-DEVICE [3], which effectively maps RDF Schema and data into objects and then reasons over RDF data using a deductive object-oriented rule language. The rule language is implemented by translating deductive rules into CLIPS production rules. Some of its features are support for incrementally maintained, materialized views, normal and generalized path expressions, stratified negation as failure, aggregate, grouping, and sorting, functions. Due to space limitations only few features of the OWL mapping scheme are presented in this paper. Furthermore, the rule language is not discussed.

The rest of the paper is organized as follows: Section 2 presents related work on rule systems on top of ontologies. Section 3 presents the overall architecture of the system, describing shortly the functionality of the basic modules of the system. Section 4 describes the mapping procedure of OWL semantics into COOL. Finally, Section 5 concludes with a summary and potential future work.

2 Related Work

A lot of effort has been made to develop rule engines for reasoning on top of OWL ontologies. Bossam [12] is a RETE-based forward chaining rule engine that a) supports both negation-as-failure and classical negation, b) relieves range-restrictedness in the rule heads and c) supports remote binding for cooperative inferencing among multiple rule engines. Bossam translates OWL documents into RDF triples as facts. Any triples referring to OWL classes and restrictions are translated into unary predicates and triples declaring property values into binary predicates. Finally, RDF collections are translated into built-in list constructs.

F-OWL [7] is an ontology inference engine for OWL, which is implemented using Flora-2, an object-oriented knowledge base language and application development platform that translates a unified language of F-logic, HiLog, and Transaction Logic into the XSB deductive engine. Key features of F-OWL include the ability to reason with the OWL ontology model, the ability to support knowledge consistency checking using axiomatic rules defined in Flora-2, and an open application programming interface (API) for Java application integrations.

ROWL [9] system enables users to frame rules in RDF/XML syntax using ontology in OWL. Using XSLT stylesheets, the rules in RDF/XML are transformed into forward-chaining rules in JESS. It makes use of two more stylesheets to transform ontology and instance files into Jess unordered facts that represent triplets. The file with facts and rules are then fed to JESS which enables inferencing and rule invocation.

SweetJess [10] is an implementation of a defeasible reasoning system (situated courteous logic programs) based on Jess that integrates well with RuleML. However, SweetJess rules can only express reasoning over ontologies expressed in DAMLRuleML (a DAML-OIL like syntax of RuleML) and not on arbitrary OWL data. Furthermore, SweetJess is restricted to simple terms (variables and atoms).

SweetProlog [13] is a system for translating rules into Prolog. This is achieved via a translation of OWL ontologies and rules expressed in OWLRuleML into a set of facts and rules in Prolog. It is implemented in Java and makes use of three languages: Prolog as a rule engine, OWL as an ontology and OWLRuleML as a rule language. It enables reasoning (through backward chaining) over OWL ontologies by rules via a translation of OWL subsets into simple Prolog predicates which a JIProlog engine can handle. There are five principle functions that characterize SweetProlog: a) translation of OWL and OWLRuleML ontologies into RDF triples, b) translation of OWL assertions into Prolog, c) translation of OWLRuleML rules into CLP, d) transformation of CLP rules into Prolog and e) interrogation of the output logic programs.

DR-Prolog [5] is a system for defeasible reasoning on the Web. The system is a) syntactically compatible with RuleML, b) features strict and defeasible rules, priorities and two kinds of negation, c) is based on a translation to logic programming with declarative semantics, and d) can reason with rules, RDF, RDF Schema and part of OWL ontologies. The system is based on Prolog and supports monotonic and nonmonotonic rules, open and closed world assumption and reasoning with inconsistencies.

SWRL [11] is a rule language based on a combination of OWL with the Unary/Binary Datalog sublanguages of RuleML. SWRL enables Horn-like rules to be combined with an OWL knowledge base. Negation is not explicitly supported by the SWRL language, but only indirectly through OWL DL (e.g. class complements). Its main purpose is to provide a formal meaning of OWL ontologies and extend OWL DL. There is a concrete implementation of SWRL, called Hoolet. Hoolet translates the ontology to a collection of axioms (based on the OWL semantics) which is then given to a first order prover for consistency checking. Hoolet has been extended to handle rules through the addition of a parser for an RDF rule syntax and an extension of the translator to handle rules, based on the semantics of SWRL rules.

SWSL [4] is a logic-based language for specifying formal characterizations of Web services concepts and descriptions of individual services. It includes two sublanguages: SWSL-FOL and SWSL-Rules. The latter is a rule-based sublanguage, which can be used both as a specification and an implementation language. It is designed to provide support for a variety of tasks that range from service profile specification to service discovery, contracting and policy specification. It is a layered language and its core consists of the pure Horn subset of SWSL-Rules.

WRL [1] is a rule-based ontology language for the Semantic Web. It is derived from the ontology component of the Web Service Modeling Language WSML. The language is located in the Semantic Web stack next to the Description Logic based Ontology language OWL. WRL constists of three variants, namely Core, Flight and Full. WRL-Core marks the common core between OWL and WRL and is thus the basic interoperability layer with OWL. WRL-Flight is based on the Datalog subset of

F-Logic, with negation-as-failure under the Perfect Model Semantics. WRL-Full is based on full Horn with negation-as-failure under the Well-Founded Semantics.

3 System Architecture

The architecture of the system (Fig 1) consists of the following five basic modules:

- Rule Program Loader
- OWL Triple Loader
- Deductive Rule Translator
- OWL Triple Translator
- OWL Extractor

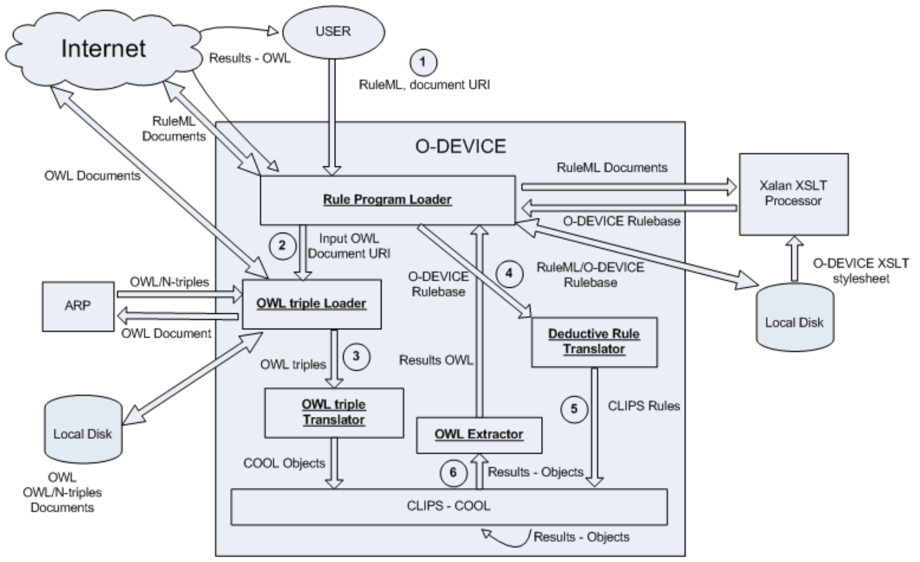

Fig. 1. Architecture of O-DEVICE

The user inputs (*step 1*) the URL of the RuleML rule file to the *Rule Program Loader*, which downloads it. The rule file also contains information about the location of the OWL files, the names of the derived classes to be exported as results and the name of the output OWL file. The *Rule Program Loader* scans the rule file to target the relevant OWL documents to which the rule file refers and passes theirs URLs to the *OWL Triple Loader* (*step 2*). The RuleML program is translated into the native O-DEVICE rule notation using an XSLT stylesheet. The O-DEVICE rule program is then forwarded to the *Deductive Rule Translator (step 4)*.

The *OWL Triple Loader* accepts a specific URL of an OWL document from the *Rule Program Loader* to download (*step 2*). Furthermore it uses the ARP Parser [14] to translate the OWL document in the N-Triple format. The *OWL Triple Translator* accepts from the *OWL Triple Loader* the produced triples (*step 3*) and transforms them into classes, properties and objects of COOL The mapping scheme is described in Section 4.

The *Deductive Rule Translator* accepts from the *Rule Program Loader* a set of O-DEVICE rules (*step 4*) and translates them into a set of CLIPS production rules (*step 5*). After the translation of deductive rules or the loading of the compiled rules, CLIPS runs the production rules and generates the objects that constitute the result of the rule program. The result-objects are exported to the user (*step 6*) as an OWL document through the *OWL Extractor*.

4 Mapping OWL Primitives into COOL

In this section we describe how the OWL data model is mapped onto the COOL object-oriented model of the CLIPS language. The class hierarchy of O-DEVICE follows precisely the class hierarchy of OWL, as it is declared in the OWL Specification [15]. The mapping of the RDF Model into COOL is already defined in R-DEVICE [3], so we just extended the existing RDF hierarchy. In this section we describe only the way O-DEVICE handles OWL primitives.

4.1 Mapping ontologies and data to objects

The mapping scheme of OWL ontologies and data to objects tries to exploit as many built-in features of the host object-oriented language (namely COOL) as possible. In this way, querying of objects and reasoning over OWL data will be faster. The main features of the basic O-DEVICE mapping scheme are the following:

- Built-in OWL classes are represented both as classes and as objects, instances of the `rdfs:Class` class. This binary representation is due to the fact that COOL does not support meta-classes, so the role of meta-class is played by the instances of `rdfs:Class` class. User-defined classes follow the same scheme except for the fact that the "meta-class" objects are instances of the class `owl:Class.` Meta-classes are needed in order to store certain information about a class. So, for example, the OWL class *Whale* (in section 4.2.1) is represented in O-DEVICE both by a `defclass Whale` construct and a `[Whale]` object that is an instance of the `owl:Class` class. Inheritance issues of class hierarchies are treated by the class-inheritance mechanism of COOL, for inheriting properties from superclasses to subclasses, for including the extensions of subclasses to the extensions of the superclasses and for the transitivity of the `rdfs:subClassOf` property.
- All OWL data (resources) are represented as COOL objects, direct or indirect instances of the `owl:Thing` class.
- Properties are direct or indirect instances of the class `owl:DatatypeProperty` or `owl:ObjectProperty`. This also includes subclasses of the above classes, such as `owl:TransitiveProperty`. Furthermore, properties are defined as slots (attributes) of their domain class(es). The values of properties are stored inside resource objects as slot values. OWL properties are multislots, i.e. they store lists of values, because a resource can have multiple times the same property attached to it.

4.2 Handling OWL Semantics

In OWL specification, there are new classes and properties that enrich the language with more semantics than RDF. For a system to be able to reason correctly on OWL documents, it should handle these classes and properties appropriately. O-DEVICE currently handles ontologies in OWL DL, which supports rich expressiveness and gives computational guarantees. In the subsections below, we describe how the system handles some of the OWL constructs, giving for each case a short example.

4.2.1 Property Restrictions

Class `owl:Restriction` is a special kind of class description. It describes an anonymous class, namely a class of all individuals that satisfy the restriction. OWL distinguishes two kinds of property restrictions: value and cardinality constraints. Value constraints are declared with the properties `owl:allValuesFrom`, `owl:someValuesFrom`, `owl:hasValue` and the cardinality constrains with the properties `owl:cardinality`, `owl:minCardinality`, `owl:maxCardinality`.

Before we explain how O-DEVICE handles these properties, we briefly describe a system slot we use to specify the type of a property range. In COOL we define the value type of a slot via the `INSTANCE-NAME` keyword for object properties and the `INTEGER`, `SYMBOL`, etc, keywords for the datatype properties. However, there is no way to specify explicitly the class that an object property is allowed to be. This is very critical for our system and we solve this problem by inserting a meta-class variable (slot), named `class-refs`, which holds the class types for all the object properties of a class. This system-slot is stored within the "meta-class" object.

Due to space limitations, below we only present how O-DEVICE handles `owl:allValuesFrom` and `owl:minCardinality`.

Restriction `owl:allValuesFrom`

A restriction containing an `owl:allValuesFrom` constraint is used to describe a class of all individuals for which all values of the property under consideration are either members of the class extension of the class description or are data values within the specified data range. Currently the system does not support data ranges. So we will give an example about the class extension of the class description.

Suppose we have the following OWL document with four classes, *Mammal, Whale, Land and Water* and one object property *living_place*. The domain of *living_place* is *Mammal* and its range is *Land*. This property is inherited by *Whale*; however, the restriction for *Whale* is that this property can take values only from *Water*.

```
<owl:Class rdf:ID="Mammal"/>
<owl:Class rdf:ID="Whale">
   <rdfs:subClassOf rdf:resource="#Mammal"/>
</owl:Class>
<owl:Class rdf:ID="Land"/>
<owl:Class rdf:ID="Water/">
<owl:ObjectProperty rdf:ID="living_place" >
   <rdfs:domain rdf:resource="#Mammal" />
    <rdfs:range rdf:resource="#Land" />
</owl:ObjectProperty>
<owl:Class rdf:about="Whale" >
```

```
   <rdfs:subClassOf>
       <owl:Restriction>
          <owl:onProperty rdf:resource="#living_place" />
          <owl:allValuesFrom rdf:resource="#Water" />
       </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

The "meta-class" object for *Whale* is shown below (without unnecessary details).

```
[Whale] of owl:Class
   (class-refs example:living_place example:Water    rdf:type owl:Class
            ...
            owl:equivalentClass owl:Class   owl:intersectionOf List)
   (rdf:type [owl:Class])
```

The multislot *class-refs* contains value pairs in the form *property-range* and holds the ranges for the values of all the object properties of a certain class. For example, the first two values of the slot are *example:living_place* and *example:Water*. This means that the object property *example:living_place* of class *example:Whale* can take values only from *example:Water*. The same slot of class *example:Mammal*, which is a super-class of *example:Whale* contains the pair *example:living_place-example:Land* taken from the `rdfs:domain` restriction of the definition of property *example:living_place*. This information is used from the system during its operation either to enforce correct data types or to infer types for values of object properties.

The `owl:hasValue` constraint is implemented as a default slot value in COOL, whereas the `owl:someValuesFrom` constraint is implemented a combination of a minimum cardinality constraint (see below) and a special run-time check upon the creation of the resource-object.

Restriction `owl:minCardinality`

This restriction defines a lower bound of the number of possible values of a property. COOL directly supports cardinality constraints for multislots. Consider the following example:

```
<owl:Class rdf:about="#Wine" >
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#madeFromGrape" />
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1
      </owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The definition of *Wine* is:

```
(defclass example:Wine    (is-a gen1)
  (multislot example:madeFromGrape (type STRING) (cardinality 1
  ?VARIABLE)))
```

The cardinality constraint has the form *(cardinality 1 ?VARIABLE)* which means that the lower bound is 1 and the upper bound unrestricted. The rest of the OWL cardinality constraints are implemented similarly.

4.2.2 Boolean Combinations of Classes

In OWL it is possible to create new classes by combining existing classes through Boolean operators. For example, the `owl:intersectionOf` property links a class to a list of class descriptions and defines the new class extension to contain precisely those individuals that are members of the class extension of all class descriptions in the list. We describe the use of this property using the following simple example:

```
<owl:Class rdf:ID="Man">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:resource="#Human"/>
    <owl:Class rdf:resource="#Male"/>
  </owl:intersectionOf>
</owl:Class>
```

Class *Man* is an intersection of classes *Human* and *Male*. This is implemented in O-DEVICE as a class that is a subclass of the other two:

```
(defclass example:Man (is-a example:Human example:Male))
(defclass example:Human (is-a owl:Thing))
(defclass example:Male (is-a owl:Thing))
```

Objects that belong to class *Man* also belong to both its superclasses through the default class extension mechanism of COOL. Furthermore, when a resource is an instance of both *Human* and *Male* classes, the core triple translator of R-DEVICE should create a dummy class, which is the intersection of the two classes, and then create a new object as an instance of this dummy class. However, if the intersection class already exists, then the new object is created as an instance of that class.

Union of classes is implemented as a common superclass. The complement of a class is implemented through the rule language of O-DEVICE by replacing all references to the complement of a class *A* with the keyword *~A*, which actually ranges over all objects that are not instances of *A*. In this way, we are able to implement the strong negation of OWL into a production rule environment where the closed world assumption holds and only negation-as-failure exists.

4.2.3 Class Equivalence

The built-in property `owl:equivalentClass` links a class description to another class description stating that both class extensions contain exactly the same set of individuals. Consider the following example:

```
<owl:Class rdf:ID="Picture" />
<owl:Class rdf:ID="Figure">
   <owl:equivalentClass rdf:resource="#Picture" />
</owl:Class>
<owl:Class rdf:ID="Image" >
   <owl:equivalentClass rdf:resource="#Picture" />
</owl:Class>
```

This is treated by creating a system class, with a random generated name, and making all the equivalent classes above subclasses of this class.

```
(defclass gen1              (is-a owl:Thing))
(defclass example:Picture   (is-a gen1))
(defclass example:Figure    (is-a gen1))
(defclass example:Image     (is-a gen1))
```

These are the corresponding "meta-objects":

```
[example:Picture] of owl:Class
   (class-refs ... owl:equivalentClass owl:Class ... )
   (owl:equivalentClass [example:Image] [example:Figure])
   (rdf:type [owl:Class])            (primary y)
   (rdfs:subClassOf [gen1])          (equivalent gen1)
[example:Image] of owl:Class
   (class-refs ... owl:equivalentClass owl:Class ... )
   (owl:equivalentClass [example:Figure] [example:Picture])
   (rdf:type [owl:Class])            (primary n)
   (rdfs:subClassOf [gen1])          (equivalent gen1)
[example:Figure] of owl:Class
   (class-refs ... owl:equivalentClass owl:Class ... )
  (owl:equivalentClass [example:Image] [example:Picture])
  (rdf:type [owl:Class])             (primary n)
  (rdfs:subClassOf [gen1])           (equivalent gen1)
```

Notice the two system properties *primary* and *equivalent*. The *primary* slot takes values '*y*' or '*n*' and is used as follows: one of the equivalent classes is randomly chosen to be a primary one (thus having the value *y* for the slot *primary*). This class is considered the representative of all the others and is used during the construction of the set of equivalent classes. Notice that the slot `owl:equivalentClass` holds the names of the rest of the equivalent classes of the set. In the example, the representative class of all the equivalent classes is *Picture*.

The *equivalent* property holds the name of the superclass of all the equivalent classes. In the example, the superclass of all equivalent classes is *gen1*. The semantics of class equivalence is actually implemented through the rule language of O-DEVICE; when a class appears in a rule, the system checks if it has a non-null value in the property *equivalent*. If yes, then the name of the class is replaced by the equivalent class. For example, if *Picture* (or *Image* or *Figure*) appears in a rule, then it will be substituted by *gen1* and thus this rule will cover all the equivalent classes since the extension of *gen1* covers the same set of objects, namely the union of the extensions of all the equivalent classes.

The `owl:sameAs` construct is handled in a similar way, by creating a primary object that represents all the "same" objects, which delegate to the primary object all their property values. In this way the primary object is always "retrieved" by the rule language, since the rest of the objects to not have any property values. When any of these objects is directly queried, it delegates the query to the primary object. In this way we overcome the unique-names assumption of OO programming languages.

4.2.4 Special Properties

In OWL several special characteristics of properties can be defined such as transitivity, symmetry, etc. For example, when a property *P* is transitive and the pairs (*x*, *y*) and (*y*, *z*) are instances of *P*, then it can be inferred that the pair (*x*, *z*) is also an instance of *P*. Consider the following example:

```
<owl:Class rdf:ID="Region" />
<owl:TransitiveProperty rdf:ID="subRegionOf">
   <rdfs:domain rdf:resource="#Region"/>
    <rdfs:range  rdf:resource="#Region"/>
</owl:TransitiveProperty>
<Region rdf:ID="region1" >
   <subRegionOf rdf:resource="#region2"/>
</Region>
```

```
<Region rdf:ID="region2" >
   <subRegionOf rdf:resource="#region3"/>
</Region>
<Region rdf:ID="region3" />
```

Region1 is a sub-region of *region2* and *region2* is a sub-region of *region3*. Because `subRegionOf` is a transitive property, the system must infer that *region1* is also a sub-region of *region3*. In O-DEVICE the corresponding objects are:

```
[example:region1] of example:Region
  (rdf:type [example:Region])   (example:subRegionOf [example:region2]
                                                     [example:region3])
[example:region2] of example:Region
  (rdf:type [example:Region])   (example:subRegionOf [example:region3])
[example:region3] of example:Region
  (rdf:type [example:Region])   (example:subRegionOf)
```

The value *example:region3* has been explicitly stored to the property `subRegionOf` of the instance *region1*. In fact, the *OWL Triple Translator* module calculates and materializes the transitive closure of transitive properties when OWL documents are loaded. In this way, during the execution of rules there is no need to navigate through transitive properties. Almost the same scheme is used to implement symmetric and inverse properties. The *OWL Triple Translator* module materializes the reverse relationships at load time. Functional and inverse functional properties are implemented with the aid of the COOL cardinality constraint.

5 Conclusions and Future Work

In this paper we have presented O-DEVICE, a deductive object-oriented knowledge base system for reasoning over OWL documents. O-DEVICE imports OWL documents into the CLIPS production rule system by transforming OWL ontologies into an object-oriented schema and instances of OWL classes into objects. In this way, when accessing multiple properties of a single resource, few joins are required. The system also features a powerful deductive rule language which supports inferencing over the transformed OWL descriptions, which however has not been presented in this paper, due to space limitations. The mapping scheme of OWL ontologies and data to COOL objects is partly based on the underlying COOL object model and partly on the compilation scheme of the deductive rule language.

O-DEVICE is still work in progress; therefore, certain features of the descriptive semantics of OWL are still under development. For example, functional properties are currently handled only as cardinality restrictions, whereas their role is also to conclude that two different names might actually represent the same resource. All these interpretations of OWL constructs are currently being implemented by appropriately extending the *OWL Triple Translator* (Fig. 1) with production rules that assert extra triples, which are further treated by the translator. Notice that asserting new properties to an already imported ontology might call for object and/or class re-definitions, which are efficiently handled by the core triple translator of R-DEVICE [3]. Therefore, the triple translator is non-monotonic, and so is the rule language, since it supports stratified negation as failure and incrementally maintained materialized views.

Planned future work includes:

- Deploying the reasoning system as a Web Service.
- Implementing a Semantic Web Service composition system using OWL-S service descriptions and user-defined service composition rules.
- Integrating O-DEVICE with a defeasible logic reasoner [2], an extension of the R-DEVICE system, and study how defeasible logic can be used to describe and implement web service composition strategies.

References

1. Angele J, Boley H., J. de Bruijn, Fensel D., Hitzler P., Kifer M., Krummenacher R., Lausen H., Polleres A., Studer R., "Web Rule Language (WRL)", Technical Report, `http://www.wsmo.org/wsml/wrl/wrl.html`
2. Bassiliades N., Antoniou G., Vlahavas I., “A Defeasible Logic Reasoner for the Semantic Web”, *3rd Int. Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML 2004)*, Springer-Verlag, LNCS 3323, pp. 49-64.
3. Bassiliades N., Vlahavas I., “R-DEVICE: A Deductive RDF Rule Language”, *3rd Int. Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML 2004)*, Springer-Verlag, LNCS 3323, pp. 65-80
4. Battle S., Bernstein A., Boley H., Grosof B., Gruninger M., Hull R., Kifer M., Martin D., McIlraith S., McGuinness D., Su J., Tabet S., "SWSL-rules: A rule language for the semantic web", *W3C rules workshop*, Washington DC, USA, April 2005
5. Bikakis A., Antoniou G., DR-Prolog: A System for Reasoning with Rules and Ontologies on the Semantic Web 2005, *Proc. 25th American National Conference on Artificial Intelligence* (AAAI-2005).
6. Boley, H., Tabet, S., and Wagner, G., “Design Rationale of RuleML: A Markup Language for Semantic Web Rules”, *Proc. Int. Semantic Web Working Symp.*, pp. 381-402, 2001.
7. Chen H., Zou Y., Kagal L., Finin T., “F-OWL: An OWL Inference Engine in Flora-2”, `http://fowl.sourceforge.net/`
8. CLIPS 6.23 Basic Programming Guide, `http://www.ghg.net/clips`
9. Gandon F. L., Sheshagiri M., Sadeh N. M., "ROWL: Rule Language in OWL and Translation Engine for JESS", `http://mycampus.sadehlab.cs.cmu.edu/public_pages/ROWL/ROWL.html`
10. Grosof B.N., Gandhe M.D., Finin T.W., “SweetJess: Translating DAMLRuleML to JESS”, *Proc. RuleML Workshop*, 2002.
11. Horrocks I., Patel-Schneider P.F., Boley H., Tabet S., Grosof B., Dean M., “SWRL: A semantic web rule language combining OWL and RuleML”, Member submission, May 2004, W3C. `http://www.w3.org/Submission/SWRL/`
12. Jang M., "Bossam - A Java-based Rule Processor for the Semantic Web", `http://mknows.etri.re.kr/bossam`
13. Laera L., Tamma V., Bench-Capon T. and Semeraro G., “SweetProlog: A System to Integrate Ontologies and Rules”, *3rd Int. Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML 2004)*, Springer-Verlag, LNCS 3323, pp. 188-193.
14. McBride B., “Jena: Implementing the RDF Model and Syntax Specification”, *Proc. 2nd Int. Workshop on the Semantic Web*, 2001
15. Web Ontology Language (OWL), `http://www.w3.org/2004/OWL/`