R-DEVICE: An Object-Oriented Knowledge Base System for RDF Metadata

Nick Bassiliades, Ioannis Vlahavas
Dept. of Informatics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
{nbassili,vlahavas}@csd.auth.gr

ABSTRACT

In this paper we present R-DEVICE, a deductive object-oriented knowledge base system for reasoning over RDF metadata. R-DEVICE imports RDF documents into the CLIPS production rule system by transforming RDF triples into COOL objects and uses a deductive rule language for reasoning about them. R-DEVICE is based on an OO RDF data model, different than the established triple-based model, which maps resources to objects and encapsulates properties inside resource objects, as traditional OO attributes. In this way, fewer joins are required to access the properties of a single resource resulting in better inferencing/querying performance, as it is experimentally shown in the paper. Furthermore, RDF can interoperate seamlessly with other web data models and languages. The descriptive semantics of RDF may call for dynamic redefinitions of resource classes, which are handled by R-DEVICE effectively. Furthermore, R-DEVICE features a powerful deductive rule language for reasoning on top of RDF metadata. The rule language includes features such as normal and generalized path expressions, stratified negation, aggregate, grouping, and sorting, functions. The rule language supports a second-order syntax, which is efficiently translated into sets of first-order logic rules using metadata, where variables can range over classes and properties, so that reasoning over the RDF schema can be made. Users can define views which are materialized and incrementally maintained by translating deductive rules into CLIPS production rules that preserve truth. Users can choose between an OPS5/CLIPS-like and a RuleML-like syntax. Finally, users can define and use functions through the CLIPS host language.

Keywords

RDF, Object Data Model, CLIPS, Descriptive Semantics, Deductive Rules, Production Rules, Generalized Path Expressions, Aggregation, Materialized Views, RuleML

1 Introduction

The Semantic Web is the next step of evolution for the World Wide Web [13], where information is given well-defined meaning, enabling computers and people to work in better cooperation. Currently, information found on the Web is mainly for human consumption and is not machine-understandable. It is quite difficult to automate things on the Web, and because of the volume of information the Web contains, it is even more difficult to manage it manually. The solution proposed by the WWW Consortium is to use metadata to describe the data contained on the Web [12]. The Resource Description Framework (RDF) is a foundation for processing metadata; it provides interoperability between applications that exchange machine-understandable information on the Web [31].

RDF is actually a general-purpose language for representing information in the World Wide Web. However, it is particularly intended for representing metadata about Web resources, such as the title, author, etc. RDF generalizes the concept of a "Web resource", so it can be used to represent information about anything that can be identified on the Web. The RDF model is based on sets of statements or triples, each of which can be thought of as a directed labelled graph in which nodes are called *resources* (or literals) and edges are called *properties*. The source node of an edge of the directed graph is the *subject* of the statement and the target is the *object*. The edge is labeled with the *predicate*. Furthermore, RDF has a schema definition language (RDFS) [15], for creating types for graph nodes (called *classes*) and edges (called *properties*). Finally, RDF has an XML syntax for expressing metadata and schemas in a form that is both human readable and machine understandable.

Conveying the content of documents is just a first step for achieving the full potential of the Semantic Web. Additionally, it is very important to be able to reason with and about information spread across the WWW, so that intelligent agents can automatically perform complicated tasks on the Web, on a user's behalf. Rules provide the natural and widely-accepted mechanism to perform automated reasoning, with mature and available theory and technology. This has been identified as a Design Issue for the Semantic Web, as clearly stated in [12].

Rules and rule mark-up languages, such as RuleML [14], will play an important role in the success of the Semantic Web. Rules will act as a means to draw inferences, to express constraints, to specify policies, to react to events/changes, to transform data, etc. Rule mark-up languages will allow enriching web ontologies by adding definitions of derived concepts, to publish rules on the web, to exchange rules between different systems and tools, etc. The applications include electronic commerce, data integration and sharing, information gathering, security access and control, law, diagnosis, B2B, and of course, to modelling of business rules and processes.

It seems natural to add rules "on top" of web ontologies. However, as it is argued in [2], putting rules and description logics together poses many problems, and may be overkill, both computationally and linguistically. Another possibility is to start with RDF/RDFS, and extend it by adding rules.

One solution to implement such a rule system is to start from scratch and build inference engines that draw conclusions directly on the RDF data model. However, such an approach tends to throw away decades of research and development of efficient and robust rule engines. In this paper we follow a different approach: we re-use an existing rule system (CLIPS [17]) for reasoning on top of RDF data. However, before an existing rule system is used, careful consideration must be given to how RDF data and semantics are going to be treated in the host system.

The semantics of the RDF data model differ from those of traditional data structures, such as the object data model, that are used in existing rule systems. Specifically, RDF is an assertional language, according to its semantics [26], i.e. each assertion declares that certain information about resources is true, including schema information, and its meaning is not changed by future assertions. This kind of semantics is called *descriptive* and is based on the *Open-World Assumption* of the Semantic Web. Traditional data models define certain constraints in their schema definitions and schema instances have to obey these constraints, i.e. no information entry that violates these constraints is allowed. This kind of semantics is called *prescriptive* and is based on the *Closed-World Assumption* of logic programming systems and databases.

A challenging task, in order to re-use existing query and inference systems with prescriptive semantics, is to be able to capture the descriptive RDF semantics in a traditional data model. In this paper, we present a deductive object-oriented knowledge base system, called R-DEVICE, which transforms RDF triples into objects and uses a deductive rule language for querying and reasoning about them.

R-DEVICE employs a novel OO-RDF model [5] that maps RDF documents into COOL objects inside the CLIPS production rule system [17]. The main difference between the RDF triple-based model and our OO-RDF model is that we treat properties mainly as attributes encapsulated inside resource objects, as in traditional OO programming languages. In this way properties about a single resource are gathered together in one object, resulting in superior inference/query performance compared to a triple-based model, as it is experimentally shown in this paper. Most other RDF inferencing/querying systems that are based on a triple model scatter resource properties across several triples and they require several joins to access the properties of a single resource. Furthermore, RDF data can interoperate seamlessly with other web data models and languages, as it is discussed in eh conclusions. The descriptive semantics of RDF data may call for dynamic redefinitions of resource classes and objects, which are handled by R-DEVICE effectively.

R-DEVICE features a powerful deductive rule language [7] which is able to draw inferences both on the RDF schema and data. The rule language includes features such as ground and generalized path expressions, stratified negation, aggregate, grouping, and sorting functions. All these can be combined with second-order syntax, where variables can range over classes and properties. Such variables are grounded at compile-time using metadata so second-order rules are safely and efficiently translated into sets of first-order rules. Furthermore, users can define views with R-DEVICE rules which are materialized and incrementally maintained by translating deductive rules into CLIPS production rules that preserve truth. Users can use built-in functions of CLIPS or can define their own arbitrary functions. The syntax of R-DEVICE rules follows the OPS5/CLIPS paradigm. Furthermore, an XML syntax is provided that extends RuleML [14] and especially the version that supports OO features and negation-as-failure.

Regarding potential applications, R-DEVICE could be used as an inference mechanism on top of an RDF repository. The RDF data would be pre-loaded into R-DEVICE and external users would submit rule programs into the system either through a form-based HTML interface or using R-DEVICE remotely as a Web-service through SOAP messaging. Changes to the base RDF metadata of the repository would be incrementally propagated to R-DEVICE, as well. Another use for R-DEVICE could be an on-the-fly RDF inferencing service, provided that the queried RDF documents are not very large, since parsing very large RDF/XML documents into triples at run-time and then importing them into R-DEVICE would not be very efficient.

In the rest of this paper we present the architecture of the system in Section 2. Section 3 describes in detail how RDF triples are mapped into CLIPS objects. Section 3.6 describes the R-DEVICE rule language. Section 5 presents some performance results of R-DEVICE queries and compares them to the results obtained by a triple-based query model of RDF, in our system. Section 6 briefly reviews related work in querying and reasoning about RDF metadata, and finally, section 7 concludes this paper and discusses future work.

2 System Architecture

The R-DEVICE system consists of the following major components (**Figure 1**):

- The Rule Program Loader
- The RDF Triple Loader
- The RDF Triple Translator
- The Deductive Rule Translator
- The RDF Extractor

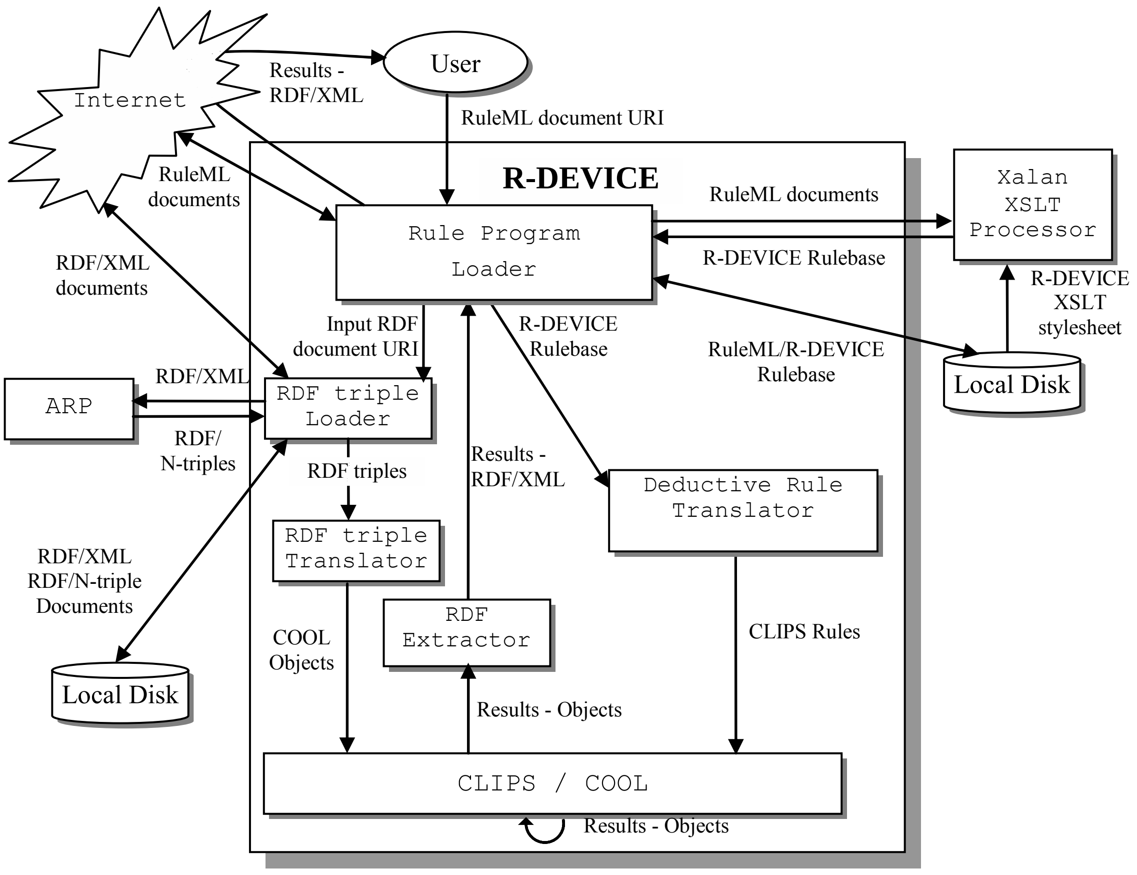


Figure 1. Architecture of the R-DEVICE system.

The *Rule Program Loader* accepts from the user a URL (or a local file name) that contains a deductive rule program in RuleML notation [14]. The RuleML document may also contain the URLs of the input RDF documents on which the rule program will run on, which is forwarded to the RDF Triple Loader. The RuleML program is translated into the native R-DEVICE rule notation using the Xalan XSLT processor [42] and an XSLT stylesheet. The R-DEVICE rule program is then forwarded to the Deductive Rule Translator.

The *RDF Triple Loader* accepts from the Rule Program Loader (or directly from the user) requests for loading specific RDF documents, downloads them from the Internet and uses the ARP parser [32] to translate them to triples in the N-triple format. Both the RDF/XML and RDF/N-triple files are stored locally for future reference. N-triples are loaded into memory. Resources that have a `URI#anchorID` or `URI/anchorID` format are transformed into a `namespace:anchorID` format, in order to save memory space. Of course, `URI` must either belong to the namespaces declared in the current RDF document or be a namespace that is already known to the system by previously loaded documents. More on namespace treatment is discussed in section 3.4.

The transformed RDF triples are fed to the *RDF Triple Translator* which maps them into COOL objects, according to the mapping schema that is described in the next section. Notice that when an RDF triple is consumed (i.e. translated) it is deleted. The loading/translation of N-Triples can be performed in either a single step or in an iterative (streaming) fashion where at each iteration only a (user-defined) fragment of the total triples is loaded/translated. It has been found experimentally (see section 5) that a fragment that leads to good overall performance and scalability is around 10,000 triples. A complete example of an RDF document and its translation into objects can be found in section 3.5.

The *Deductive Rule Translator* accepts from the Rule Program Loader a set of R-DEVICE rules and translates them into a set of CLIPS production rules. Details about the translation scheme are given in section 4.2. Compiled rules are kept in local files, so that the next time they are needed they can be directly loaded, increasing speed. After the translation of deductive rules or the loading of compiled rules, CLIPS runs the production rules and generates the objects that constitute the result of the initial rule program. Finally, the result-objects are exported to the user as an RDF/XML document through the *RDF Extractor*. An example of the results produced by an R-DEVICE rule can be found in section 4.3.

3 The Object-Oriented RDF Model

In this section we describe how the RDF data model is mapped onto the COOL object-oriented model of the CLIPS language [17]. Figure 2 shows the top levels of the class hierarchy of R-DEVICE. Class `USER` is a system-defined class in COOL, which serves as the root of the user-defined class hierarchies. Class `RDF-CLASS` (also considered system-defined) is the root of the R-DEVICE classes, which defines system slots and methods shared by all RDF classes (Figure 3). The main features of this mapping scheme are discussed in the following subsections.

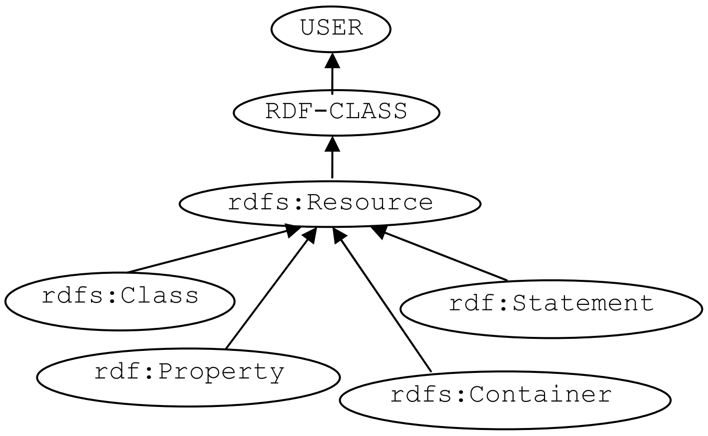


Figure 2. The class hierarchy of R-DEVICE.

3.1 Resources and Classes

RDF Schema *classes* are represented both as COOL classes and as (direct or indirect) instances of the `rdfs:Class` class. This dual representation is due to the fact that COOL does not support meta-classes, so the role of meta-classes is played by the `rdfs:Class` class, whose instances are considered as meta-classes of the corresponding classes. Class names follow the `namespace:anchorID` format, if their URI can be resolved in this way, while their corresponding instances have an object identifier with the same name, surrounded by square brackets. Figure 3 shows the definition for `rdfs:Resource` both as a class (subclass of `RDF-CLASS`) and as an instance of class `rdfs:Class`. Notice that when a resource class does not have an explicit superclass it is made a subclass of `rdfs:Resource` (entailment rule *rdfs8* in [26]).

Inheritance issues of class hierarchies are not explicitly treated in R-DEVICE, since we rely on 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 (entailment rule *rdfs9*) and for the transitivity property of the `rdfs:subClassOf` property (entailment rule *rdfs11*).

```
(defclass RDF-CLASS
   (is-a USER)
   (role concrete)  (pattern-match reactive)
   (slot uri (type STRING))
   (slot source (type SYMBOL) (default rdf))
)

(defclass rdfs:Resource
   (is-a RDF-CLASS)
   (multislot rdfs:isDefinedBy (type INSTANCE-NAME))
   (multislot rdf:type (type INSTANCE-NAME))
   (multislot rdf:value)
   (multislot rdfs:comment (type LEXEME))
   (multislot rdfs:label (type LEXEME))
   (multislot rdfs:seeAlso (type INSTANCE-NAME))
)

(definstances rdf_classes
    ...
   ([rdfs] of rdfs:Resource
      (rdfs:isDefinedBy [rdfs])
      (rdf:type [rdfs:Resource])
      (uri "http://www.w3.org/2000/01/rdf-schema#")
      (rdfs:comment "RDF Schema Vocabulary namespace")
      (rdfs:label rdfs)
      (rdfs:seeAlso [rdfs-more])
   )
    ...
   ([rdfs:Resource] of rdfs:Class
      (rdfs:isDefinedBy [rdfs])
      (rdf:type [rdfs:Class])
      (rdfs:label Resource)
      (rdfs:comment "The class resource")
      (class-refs   rdfs:isDefinedBy rdfs:Resource
                    rdf:type rdfs:Class
                    rdfs:seeAlso rdfs:Resource)
      (aliases      rdfs:seeAlso rdfs:isDefinedBy)
   )
    ...
   ([rdfs:isDefinedBy] of rdf:Property
      (rdfs:isDefinedBy [rdfs])
      (rdf:type [rdf:Property])
      (rdfs:domain [rdfs:Resource])
      (rdfs:range [rdfs:Resource])
      (rdfs:subPropertyOf [rdfs:seeAlso])
      (rdfs:label isDefinedBy)
      (rdfs:comment "Namespace of a resource")
   )
    ...
)
```

Figure 3. Various COOL definitions of RDF elements.

All *resources* are represented as COOL objects, which are (direct or indirect) instances of the `rdfs:Resource` class. The identifier of a resource object is either in `namespace:anchorID` format, if its URI can be resolved in this way, or its complete address otherwise. R-DEVICE also represents the documents of the namespaces as resource objects, storing their URI in the `uri` slot. Figure 3 shows the definition of the `rdfs` namespace as a resource object.

Blank nodes are handled as first-class resources. However, since the ARP parser guarantees the uniqueness of their names only within each document, R-DEVICE has an extra mechanism during the loading of a document that augments the name of each blank node with a unique file ID. In this way, blank nodes a globally have unique ID within R-DEVICE. Notice that the semantics of each type of blank node (e.g. lists, sequences, stratified statements, etc.) are left to the user who provides appropriate rules for interpreting them. However, the processing of such RDF constructs can benefit from certain optimizations within the CLIPS system. For example, lists and containers could be condensed into multislots for faster access. Such optimizations are currently implemented into an OWL-aware extension of R-DEVICE [34].

The specific class of each resource object depends on the `rdf:type` property of the resource. When a resource has multiple `rdf:type` properties then the resource object belongs to multiple classes. This cannot be handled directly in COOL (and in most object-oriented programming languages), therefore a dummy class is generated which is a subclass of all the classes that the object should belong to. Then the resource object is made an instance of this class. The slot `source` indicates whether an object is a proper RDF resource or a system-generated object. For example, consider the following RDF triples that define a resource `ex:A` with two types:

```
ex:A rdf:type ex:P .
ex:A rdf:type ex:Q .
```

R-DEVICE will create a dummy class `gen1` that is a subclass of classes `ex:Q` and `ex:P` as Figure 4 shows. Object `ex:A` becomes an instance of this class; however, in the slot `rdf:type` of this object the original types are kept and not the system-generated class. Furthermore, the meta-data of class `gen1` (instance `[gen1]` of class `rdfs:Class`) records in slot `system` that this is a system-generated class.

```
(defclass gen1
   (is-a ex:Q ex:P)
   ...
)
([gen1] of rdfs:Class
   (source system)
   (rdf:type [gen1])
)
([ex:A] of gen1
   (source rdf)
   (rdf:type [ex:Q] [ex:P])
)
```

Figure 4. Example of a resource with two types in R-DEVICE.

3.2 Properties

Properties are (direct or indirect) instances of the `rdf:Property` class. Furthermore, properties whose domain is a single class are defined as slots (attributes) of this class. Figure 3 shows property `rdfs:isDefinedBy` whose domain is class `rdfs:Resource`. The values of properties are stored inside resource objects as slot values. Actually, RDF properties are multislots, i.e. they store lists of values, because a resource can have multiple (different) values for the same property.

When a property has multiple domains, then a dummy class is generated which is a subclass of all the classes of the property domain. The property is then made a slot of this dummy class, since resource objects that have this property must be instances of all the classes in the domain. For example, the following set of triples defines a new property `ex:Property1` with two domains, classes `ex:P` and `ex:Q`. Figure 5 shows how a system-generated class `gen1` is created in order to "host" this property.

```
ex:P rdf:type rdfs:Class .
ex:Q rdf:type rdfs:Class .
ex:Property1 rdf:type rdf:Property .
ex:Property1 rdfs:domain ex:P .
ex:Property1 rdfs:domain ex:Q .
```

Properties with no domain constraint must be attached to all resource objects, therefore they should become slots of the `rdfs:Resource` class, which is the root of the resource object hierarchy. However, the `rdfs:Resource` class is already defined by the system, which means that it should be dynamically re-defined. This is due to RDF descriptive semantics which may add new properties to already existing classes and is treated in R-DEVICE by re-defining classes at run-time, which is discussed in section 3.3.7.

```
([ex:Property1] of rdf:Property
   (rdf:type [rdf:Property])
   (rdfs:domain [gen1])
)
([gen1] of rdfs:Class
   (source system)
   (rdf:type [gen1])
)
(defclass gen1
   (is-a ex:P ex:Q)
   (multislot ex:Property1)
   ...
)
```

Figure 5. Example of a property with two domains in R DEVICE.

Similar dynamic re-definitions occur on a couple other occasions in R-DEVICE, since new triples can be incrementally added in the knowledge base. For example, when an existing class gets a new `rdfs:subClassOf` property, then the above dynamic class re-definition occurs to alter the class hierarchy. Another case is when an existing object is given a new `rdf:type` property then the object is deleted and re-created under the scheme of multiple types described above. One reason for giving a new `rdf:type` property to an existing object is when a property is attached to an object, but the object does not belong to the domain (or the range) of the property. According to the descriptive semantics of RDF Schema (entailment rules *rdfs2*and *rdfs3* in [26]), the type of this object should be the domain (or range) of the property. For example, consider the following RDF test case 3 of the rdf-containers-syntax-vs-schema issue[1]. The `_:bar` resource has type `foo:Bar` according to the first triple. In addition, the same resource must also be of type `rdfs:Container` because this is the domain of `rdf:_XX` properties.

```
_:bar rdf:type foo:Bar .
_:bar rdf:_1 "1" .
_:bar rdf:_2 "2" .
```

Generally speaking, R-DEVICE does not reject any RDF triple because every asserted triple is considered to be true. Notice that this behaviour is compatible with the Open-World Assumption of RDF and Semantic Web in general, where every statement is considered to be true. Under the Closed-World Assumption some statements

[1] `http://www.w3.org/2000/03/rdf-tracking/#rdf-containers-syntax-vs-schema`

would cause a consistency violation error. Such behaviour could be very easily implemented in R-DEVICE, since production rule environments like CLIPS are based on the Closed-World Assumption. However, this was a design choice for R-DEVICE. An alternative would be to leave on the user the choice on which assumption to base the RDF transformation.

The `rdfs:range` constraint of properties defines the type of the values of slots. Specifically, when this constraint is absent, then there is no type constraint for the slots, while if the value of the constraint is `rdfs:Literal` then the corresponding slot is of type `STRING` or `SYMBOL` (called `LEXEME` in COOL). Furthermore, we have catered for mapping some of the XML Schema data types to the COOL data types, through the `rdfs:Datatype` class. More specifically, `xsd:integer`, `xsd:long`, etc. are casted to `INTEGER`, `xsd:float`, `xsd:decimal`, etc. are casted to `FLOAT`, while all other data types are treated as either CLIPS strings or symbols. Each of these datatypes is automatically made a subclass of `rdfs:Literal` (entailment rule *rdfs13* in [26]). However, notice that datatype classes are not real classes, in the sense that they do not have instances, but they are mere instances of `rdfs:Class`.

When the value of the range constraint is the name of a resource class (object property), the type of the slot is `INSTANCE`, the COOL datatype for object referencing slots, i.e. slot values are OIDs of resource objects. When there are multiple range constraints, R-DEVICE creates a dummy class (similarly to the case of multiple domain constraints) which becomes the type of the slot. In COOL, reference slots cannot be constrained to take as values instances of a specific class, therefore the referenced class information must be kept via a separate mechanism. Actually, since classes are not first-class objects in COOL (e.g. they cannot have class variables) we had to devise a mechanism to keep the types of reference slots. This was achieved in COOL through a multislot, called `class-refs`, which is kept inside the meta-class of each class, which is actually the `rdfs:Class` instance of each class (Figure 3). Values in this slot come in pairs; the first member of the pair is the name of a reference slot, while the second is the type (class) of the reference slot. Each class in the hierarchy has its own value. When a new class is created, it inherits the value of this slot of its superclass and adds the slot-class pairs for its own slots.

Finally, when there is no range constraint, the property can have as a value anything, i.e. either a resource or a literal. The corresponding slot definition in COOL simple does not have a type constraint.

Lastly, we discuss how property hierarchies are treated in R-DEVICE. When property A is a subproperty of B, then property B can be used wherever property A can be used, but not vice-versa (entailment rule *rdfs7* in [26]). Therefore, we can consider that property B can be an *alias* for property A. The aliasing mechanism is implemented in a way similar to the class references described above. Specifically, there is an `alias` multislot kept inside the meta-class (Figure 3). Values inside this slot come in pairs; the first member of the pair is the name of a superproperty while the second is the name of the subproperty. Actually, this slot contains the explicit transitive closure of the property hierarchy, since a property is transitively a subproperty of all the properties in the property hierarchy (entailment rule *rdfs5*). When a property is about to be added as a slot in a class, the property hierarchy is navigated upwards and all the superproperties are added in the `alias` slot as aliases of the new property. Furthermore, when a new class is created the value of the `alias` slot of its superclass(es) is inherited.

Aliases are used by the deductive rule compiler to augment a rule base with rules that refer to superproperties (see section 4.2).

3.3 The RDF Triple Translator

In this subsection we present the RDF Triple Translator in more detail. The RDF Triple Translator is implemented as a CLIPS production rule program. Some production rules consume RDF triples and create COOL resource objects, filling up their slots with properties, while other rules examine these resource objects and enforce RDF model theory, i.e. they create COOL classes and they treat property hierarchies using the aliasing mechanism. Actually, the translator rule base is clustered into rule groups that treat several aspects of the translation that have been mentioned above. Figure 6 illustrates the workflow among the various rule groups and subgroups of the RDF translator, which are described in detail in the following sub-subsections.

```
Repeat - Until no untreated triples exist
     Repeat - Until no more RDF triples are consumed
          1. Create resource objects
            1.1  Create Objects with single type
            1.2  Create Objects with multiple types
            1.3  Change type of existing object
          2. Put slot values
            2.1  Assert new type for existing object
          3. Inherit property domains/ranges
          4. Treat properties with multiple domains/ranges
            4.1  Create dummy classes
          5. Create classes
            5.1  Insert superclasses
            5.2  Insert slot definitions
            5.3  Treat property hierarchy (aliasing)
     6. Prepare re-definitions of existing classes
            6.1  Find new superclasses for existing classes
            6.2  Find new properties for existing classes
     If there exist classes that need re-definition then
          For each class that needs re-definition
               7. Re-define class
                 7.1  Backup class
                      7.1.1    Backup subclasses
                      7.1.2    Backup instances
                 7.2  Undefine class
                      7.2.1    Undefine subclasses
                      7.2.2    Delete instances
                      7.2.3    Undefine class definition
                 7.3  Redefine class
                      7.3.1    Insert new superclasses
                      7.3.2    Insert new slots
                      7.3.3    Redefine class definition
                      7.3.4    Restore subclasses
                      7.3.5    Restore instances
          Put slot values (same as step 2)
     8. Treat remaining triples
       8.1  Generate container membership properties
       8.2  Assert type of non-existing properties
       8.3  Assert type of non-existing triple subjects
            8.3.1  Assert new type for existing subjects
       8.4  Assert type of non-existing triple objects
```

Figure 6. RDF triple translator algorithm.

3.3.1. Create Resource Objects

When a triple (*Resource1 rdf:type Resource2*) is found, then it means that the object *Resource1* of class *Resource2* must be created. However, before this is done, the following things must be checked first:

- If the class *Resource2* does not yet exist, then the creation of object *Resource1* must be postponed until this is done.

- If the object *Resource1* already exists, then this means that a new triple has been added that gives the object a new (extra) type. This is treated by re-creating the object under a class that is subclass of both of its current and new classes. If this class does not yet exist, it is created on-the fly.

Notice that the object *Resource1* is not immediately created when the above triple is found, because there may exist more such triples, giving *Resource1* more types. Instead, a temporary construct is created that gathers all types for *Resource1*. When no more such types exist, then the object is actually created. The treatment of single- versus multi-type resource objects has been described in subsection 3.1.

3.3.2. Put Slot Values

When a triple (*Resource Property Value*) is found, then if the object *Resource* has already been created, its slot *Property* can be filled with the value *Value*. If object *Resource* does not yet exist, then the fill-up of the slot is postponed. If slot *Property* does not yet exist in the class of *Resource*, then the fill-up of the slot is postponed, until the class is re-defined (see section 3.3.6). Multi-slots have already been discussed. Before the value is inserted, it is checked if it is already there.

If *Value* is a literal then it is transformed to an appropriate CLIPS data type, as explained in section 3.2. On the other hand, if *Value* is a resource then the object *Value* must have already been created, otherwise referential integrity is at risk. If object *Value* does not exist, then the fill-up of the slot is postponed. Finally, if object *Value* belongs to a type (class) that is not compatible with the range restrictions of *Property*, then the fill-up of the slot is postponed and a new triple that asserts a new (extra) type for object *Value* is created. This new type will be handled by the "*Create Resource Objects*" task (see section 0) in the next cycle of the algorithm: the object *Value* will be re-created with an extra type and the slot will finally be filled-up with *Value*.

3.3.3. Inherit Property Domains/Ranges

Inheritance for property hierarchies is not treated by COOL (as it is for class hierarchies), therefore an explicit treatment is needed. The inheritance algorithm is straightforward: for each superproperty-subproperty pair each domain and range of the superproperty is "copied" to the subproperty (entailment rules *ext3*, *ext4* in [26]). Notice that the domains and ranges of the properties are explicitly stored at the corresponding property objects. The treatment of the properties as slots of classes is explained in the "*Create Classes*" task (section 3.3.5).

3.3.4. Treat Properties with Multiple Domains/Ranges

After domain/range property inheritance has been performed, properties with multiples domains/ranges are treated, according to the scheme described in section 3.2. Specifically, system classes are generated which are subclasses of the classes of the multiple domains/ranges. These new classes (in the case of multiple domains) host the properties with the multiple domains. Notice that new classes are created only if they do not already exist. Furthermore, if inside the set of multiple classes some classes are subsumed by others also in the set, then only the most specific classes remain in the set.

3.3.5. Create Classes

The step of class creation does not rely on triple consumption but on instances of `rdfs:Class`, which have been created by step "*Create Resource Objects*" (section 0) and filled-up with property values by step "*Put Slot*

Values" (section 3.3.2). All such instances that do not yet correspond to classes and have superclasses that have already been created are candidate classes for creation. This creation, however, is performed in two steps: in the first step all the necessary information for the new class is collected in a temporary construct, and in the second step this construct is used to create the new class.

The information collected for the new class includes the names of its direct superclasses and the names/types of its properties/slots. For each of the superclasses, the aliases and reference types of the inherited properties (see section 3.2) are recursively collected and locally kept. Furthermore, for each new property of the new class the correct slot definition is created that includes the name of the property and its COOL datatype, based on the range restriction (section 3.2). In the case of a resource property, the class(-es) of the range restriction are kept along with the inherited reference property types. Finally, if the new property does have superproperty(-ies) the property inheritance mechanism keeps the class aliases consistent.

Notice that the phase of class creation should normally proceed the phase of resource creation (section 3.3.1). However, in R-DEVICE this is the other way around, simply because even normal user-defined RDF classes are themselves first-class objects, instances of `rdfs:Class`. Therefore, in order to iterate over the instances of `rdfs:Class` and create the appropriate COOL classes, these instances must first have been created. Therefore, during the first iteration of the internal loop of Figure 6 only the user-defined classes will be created; the instances of these classes will be created during the second iteration.

If the RDF Schema contains further levels of meta-classes, more iteration will be needed. For example, the following triples define a meta-class `ex:mc` which is a subclass of `rdfs:Class`. Then it defines class `ex:A` as an instance of this meta-class. Finally, it defines resource `ex:aaa` as an instance of class `ex:A`. During the first iteration the object `[ex:mc]` and the class `ex:mc` will be created. During the second iteration the object `[ex:A]` and the class `ex:A` will be created. Finally, in the third iteration the object `[ex:aaa]` will be created.

```
ex:mc rdf:type rdfs:Class
ex:mc rdfs:subClassOf rdfs:Class
ex:A rdf:type ex:mc
ex:aaa rdf:type ex:A
```

This treatment of meta-classes is achieved because rules of this phase treat both direct and indirect instances of `rdfs:Class` similarly.

3.3.6. Prepare Re-definitions of Existing Classes

When no more triples can be consumed this usually means that existing objects cannot store new property values simply because their classes do not have the appropriate slots. This calls for a re-definition of the class schema, as discussed in section 3.2. The actual re-definition process is described in section 3.3.7. In this sub-subsection we describe how R-DEVICE prepares the re-definition by collecting appropriate information about new properties and/or superclasses of existing classes. Actually the class definition is backed-up in a temporary construct that holds the name of the class, the names-types of its existing slots, its current superclasses, the reference types of its properties (including inherited ones), and the property hierarchy (aliases) of properties (including inherited ones).

For each new property, the above construct is augmented with the appropriate information for the new property, such as name, type, alias, etc., in a way very similar to the actual "*Create Classes*" task (section 3.3.5). Furthermore, each new superclass is also kept in the construct taking into account class subsumption. Finally, for each new superclass, the aliases and reference types of the inherited properties are recursively collected and augment the current class definition.

Here we notice that after class re-definition a "*Put Slot Values*" task is re-performed in order to fill up the newly created properties of classes-objects with values and consume the remaining triples.

3.3.7. Dynamic Class Re-definition

Dynamic class re-definition occurs only if the previous step decides it is needed and requires the following steps:

- Back-up to a file all instances of the re-defined class.
  - Recursively back-up all the subclasses of the re-defined class.
- Back-up to a main-memory construct the definition of the class and its subclasses. These definitions are needed to re-build the classes later. The information to be backed-up has been described in section 3.3.6.
- Delete all instances of the re-defined class.
  - Recursively delete all instances of all the subclasses of the re-defined class.
- Un-define the re-defined class.
  - Recursively un-define all the subclasses of the re-defined class (in a bottom-up manner).
- Re-define the class.
  - Add the new superclass(-es) and the new property(-ies). Actually, augment the class definition that was backed-up in a main-memory construct (see above) and re-build the class using the augmented definition. The process has been described in section 3.3.6.
  - Recursively re-create all the subclasses of the re-defined class (in a top-down manner). Use the backed-up in main-memory constructs definitions.
  - Re-insert the backed-up instances into the re-defined class and all its subclasses.

Due to the fact that multiple dynamic class re-definitions can be costly, R-DEVICE tries to keep them to a minimum by gathering together all such needed re-definitions and postponing their application until the triple loading/translation cannot continue without re-defining the schema (see Figure 6). A schema import mechanism that is in conformance to the above principle is also described in section 3.4. In section 5 we include a performance comparison between having and not having dynamic class re-definition which shows that schema re-definition does not incur a very high overhead as a fraction of the total triple import time.

3.3.8. Treat Remaining Triples

Finally, when all the above tasks have been performed and unconsumed triples still exist one of the following things happens:

- A triple (*S P O*) cannot be consumed because its subject *S* cannot be created as a COOL object. This happens when the subject *S* does not have a type. In this case the domain *D* of the property *P* is assumed to be the type of *S* and a new triple (*S rdf:type D*) is asserted that makes this assumption explicit (entailment rule *rdfs2* in [26]). If property *P* does not have a domain restriction, then *S* is assumed to be of type `rdfs:Resource` (entailment rule *rdfs4a*). Notice that when a property has multiple domains, then resource *S* will have multiple types (see section 3.1). Furthermore, if multiple such triples for the same resource *S* exist, then the set of the types of *S* will be the union of the domains of the properties of all such triples.

- Exactly the same situation occurs when the object *O* of the triple cannot be created. In this case the range restriction is used to assert the type of the resource (entailment rules *rdfs3* and *rdfs4b*).
- A triple (*S P O*) cannot be consumed when its subject *S* does not have a slot named *P*. This can happen when the domain restriction *D* of *P* is incompatible with the type of *S*. In this case, *S* is assumed to have a new extra type, i.e. the domain restriction of the property and the triple (*S rdf:type D*) is asserted (entailment rule *rdfs2*). Under the Closed-World Assumption this case would cause a consistency violation error. However, in R-DEVICE we have chosen to stick with the Open-World Assumption, as already discussed.
- A triple (*S P O*) cannot be consumed when its property *P* does not exist at all as a COOL object, therefore it could not be possibly be a slot of any class. This can happen when there is no triple that asserts that the certain resource *P* is of type `rdf:Property`; therefore, the triple (*P rdf:type rdf:Property*) is asserted (entailment rule *rdf1*). Furthermore, if the object *O* has a literal datatype *DT* (i.e. *O* ≡ *Val* ^^ *DT*), then datatype *DT* is assumed to be the range constraint of the property and a new triple (*P rdfs:range DT*) is asserted. This is not explicitly stated in [26], but we believe it is a reasonable assumption.
- A special situation similar to the above occurs when the property is of `rdf:_XX` type (container-membership property). In this case two triples are asserted: the first (*rdf:_XX rdf:type rdfs:ContainerMembershipProperty*) states that the property is of type `rdfs:ContainerMembershipProperty` (RDFS axiomatic triples in [26]), and the second (*rdf:_XX rdfs:subPropertyOf rdfs:member*) that the property is a subproperty of `rdfs:member` (entailment rule *rdfs12*).

After all the above triples are checked, the triple translation algorithm starts from the beginning and continues until all the triples both original and asserted have been consumed.

3.4 RDF Schema Import

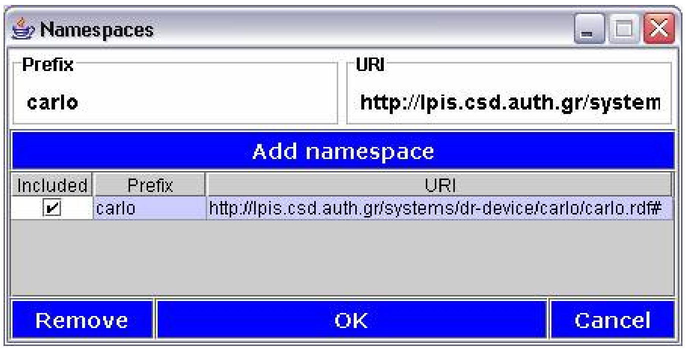
RDF documents usually refer to existing RDF Schema documents through the namespace mechanism. Although the semantics of an RDF document is precisely defined by the RDF semantics [26], the knowledge of the RDF Schema that an RDF document follows provides better understanding of its content both for the human and the machine (reasoning engine). Furthermore, the knowledge of the schema allows for more knowledgeable (and thus more efficient) rules/queries over the contents of the RDF document.

For all the above reasons, we argue that when an RDF document is loaded into R-DEVICE it is preferable to also load its schema. For this reason, RDF documents are scanned for namespaces that have not already been imported/translated into the system. Some of the untranslated namespaces may already exist on the local disk, while others are fetched from the Internet. All namespaces (both fetched and locally existing) are recursively scanned for namespaces, which are also fetched if not locally stored. Finally, all untranslated namespaces are parsed using the ARP parser and are loaded and imported into R-DEVICE using the mechanisms described in this section. Notice that the RDF Schema import phase precedes RDF document import phase.

The rationale for recursively translating all namespaces is to minimize the number of OO schema redefinitions. Fetching multiple RDF schema files will aggregate multiple RDF-to-OO schema translations into a single OO schema redefinition. Namespace resolution is not guaranteed to yield an RDF schema document; therefore, if the namespace URI is not an RDF document, then the ARP parser will not produce triples and R-DEVICE will make assumptions, based on the RDF semantics, about non-resolved properties, resources, classes, etc.

Notice that the scheme we use is "nondeterministic", because if a resource is temporarily unavailable, then when loading an RDF document its namespace (i.e. RDF Schema) will not be fetched and the RDF descriptions will be translated differently than if the schema were available. However, we find that this "nondeterminism" is compatible with the unstable nature of the Web.

Of course, the above automatic namespace/schema handling may not be desired by the user. In this case, the feature can be simply turned off. Users can then manually and explicitly import RDF Schema documents before the loading of the actual RDF instance documents. This explicit import of a schema is not available in RDF/S, but the issue is resolved in OWL [33], where explicit import of remote ontologies is supported. Furthermore, we are currently developing an integrated visual rule-base development environment for R-DEVICE [4], where the user can exert very fine control over the imported RDF ontologies, by explicitly selecting only those that he/she wishes to import. The development environment employs a similar "namespace hunting" mechanism where the user is prompted with all the collected namespaces and selects only the desired ones (Figure 7).



Prefix: carlo
URI: http://lpis.csd.auth.gr/system

Included	Prefix	URI
✓	carlo	http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo.rdf#

Figure 7. Namespace selection.

3.5 Example of RDF Document Import

Here we present a complete example of loading/translating an RDF document into R-DEVICE. Figure 8 shows an example of an RDF/XML document from the ODP metadata (see section 5) that is imported into R-DEVICE with the interaction shown in Appendix D. Notice that the Dublin Core namespace is assumed to be already loaded into the system. Figure 9 shows the classes that correspond to this RDF document, namely `dmoz:Topic` and `dmoz:ExternalPage`. Furthermore, class `rdfs:Resource` had to be re-defined (Figure 9) to add new properties, such as `dmoz:narrow` and `dmoz:link`, which do not have a specific domain, therefore their default domain is `rdfs:Resource`. Finally, Figure 10 shows all the objects that are created in R-DEVICE specifically for the RDF document of Figure 8. The RDF Extractor is described after the R-DEVICE rule language in section 4.3, because its description depends on the understanding of the deductive rule syntax and semantics.

```
<!DOCTYPE rdf:RDF [
  <!ENTITY dmoz "http://directory.mozilla.org/rdf/">
]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:dmoz="&dmoz;">
  <dmoz:Topic rdf:about="&dmoz;Top">
```

```
    <dmoz:catid>1</dmoz:catid>
    <dc:title>Top</dc:title>
    <dmoz:narrow rdf:resource="&dmoz;Top/Arts"/>
  </dmoz:Topic>
  <dmoz:Topic rdf:about="&dmoz;Top/Arts">
    <dmoz:catid>2</dmoz:catid>
    <dc:title>Arts</dc:title>
    <dmoz:link rdf:resource="http://www3.bc.sympatico.ca/PHILLIPSHOTGLASS/GlassPage.html"/>
  </dmoz:Topic>
  <dmoz:ExternalPage rdf:about="http://www3.bc.sympatico.ca/PHILLIPSHOTGLASS/GlassPage.html">
    <dc:title>John Phillips Blown glass</dc:title>
    <dc:description>A small display of glass by John Phillips</dc:description>
  </dmoz:ExternalPage>
</rdf:RDF>
```

Figure 8. Sample RDF/XML document.

```
(defclass rdfs:Resource                           (defclass dmoz:ExternalPage
   (is-a RDF-CLASS)                                  (is-a rdfs:Resource)
   (multislot dmoz:narrow)                        )
   (multislot dmoz:catid)
   (multislot dmoz:link)                          (defclass dmoz:Topic
   (multislot dc:title)                              (is-a rdfs:Resource)
   ...                                            )
   (multislot rdf:type (type INSTANCE-NAME))
   ...
)
```

Figure 9. R-DEVICE classes that correspond to the RDF document of Figure 8.

```
([dmoz] of rdfs:Resource                        ([dmoz:narrow] of rdf:Property
   (uri "http://directory.mozilla.org/rdf/")       (source rdf)
   (source system)                                 (rdf:type [rdf:Property])
   (rdfs:isDefinedBy [dmoz])                       (rdfs:domain)
   (rdf:type [rdfs:Resource])                      (rdfs:range)
   (rdfs:label dmoz)                               (rdfs:subPropertyOf)
)                                               )
([dmoz:catid] of rdf:Property                   ([dmoz:link] of rdf:Property
   (source rdf)                                    (source rdf)
   (rdf:type [rdf:Property])                       (rdf:type [rdf:Property])
   (rdfs:domain)                                   (rdfs:domain)
   (rdfs:range)                                    (rdfs:range)
   (rdfs:subPropertyOf)                            (rdfs:subPropertyOf)
)                                               )
([dmoz:Topic] of rdfs:Class                     ([dmoz:ExternalPage] of rdfs:Class
   (source rdf)                                    (source rdf)
   (rdf:type [rdfs:Class])                         (rdf:type [rdfs:Class])
   (rdfs:subClassOf)                               (rdfs:subClassOf)
)                                               )
([dmoz:Top] of dmoz:Topic                       ([dmoz:Top/Arts] of dmoz:Topic
   (source rdf)                                    (source rdf)
   (dmoz:narrow [dmoz:Top/Arts])                   (dmoz:narrow)
   (dmoz:catid "1")                                (dmoz:catid "2")
   (dmoz:link)                                     (dmoz:link [http://www3.../GlassPage.html])
   (dc:title "Top")                                (dc:title "Arts")
   (rdf:type [dmoz:Topic])                         (rdf:type [dmoz:Topic])
)                                               )
([http://www3.../GlassPage.html] of dmoz:ExternalPage
   (source rdf)
   (dmoz:narrow)
   (dmoz:catid)
   (dmoz:link)
   (dc:description "A small display of glass by John Phillips")
   (dc:title "John phillips Blown glass")
   (rdf:type [dmoz:ExternalPage])
)
```

Figure 10. R-DEVICE objects that correspond to the RDF document of Figure 8.

3.6 Completeness of the Translation

In this sub-section we discuss completeness issues of the RDF-to-object translation scheme of R-DEVICE. We claim that R-DEVICE is complete with respect to both RDF and RDFS reasoning because it covers all of the RDF and RDFS entailment rules in [26]. Furthermore, we have tested the system with the W3C RDF Test Cases [22] and it can currently handle all the approved ones. Table 1 summarizes how R-DEVICE treats the RDF and

RDFS entailment rules by referring to the appropriate sections of the paper where each entailment rule is discussed.

Table 1. Treatment of RDF/RDFS entailment rules in R-DEVICE.

Entailment Rule	Treatment in R-DEVICE	Entailment Rule	Treatment in R-DEVICE
rdf1	section 3.3.8	rdfs7	sections 3.2, 4.2
rdf2	Similar to simple entailment rules.	rdfs8	section 3.1
rdfs1	Similar to simple entailment rules.	rdfs9	section 3.1
rdfs2	sections 3.2, 3.3.8	rdfs10	Trivial. Handled by production rule language.
rdfs3	sections 3.2, 3.3.8	rdfs11	section 3.1
rdfs4a	section 3.3.8	rdfs12	section 3.3.8
rdfs4b	section 3.3.8	rdfs13	section 3.2
rdfs5	section 3.2	ext3	section 3.3.3
rdfs6	Trivial. Handled by production rule language.	ext4	section 3.3.3

Simple entailment rules, such as generalization-instantiation rules and datatype entailment rules, are trivial and they are implicitly handled by the semantics of the object-oriented language of CLIPS (COOL) and by the production rule language of CLIPS. This is also true for entailment rules *rdf2* and *rdfs1* in Table 1, which complement literal generalization rules. Furthermore, entailment rules *rdfs6* and *rdfs10*, which suggest that sub-property and subclass relations are reflective, are also trivial and they are implicitly handled by the semantics of the production rule language of CLIPS. Finally, R-DEVICE supports extensional entailment rules *ext3* and *ext4* which suggest that domains and ranges of properties are inherited by their sub-properties.

4 The Rule Language

R-DEVICE belongs to the family of deductive object-oriented rule languages ([6], [8], [10], [9]). There are three types of rules in R-DEVICE: deductive rules, derived attribute rules and aggregate attribute rules. In the following subsections we present the syntax and semantics of R-DEVICE rules (section 4.1) and the translation of R-DEVICE rules in CLIPS production rules (section 4.2). Finally, in section 4.3 we present the extraction of the results of the R-DEVICE reasoning process as an RDF document.

4.1 The Rule Syntax

The syntax of R-DEVICE rules is a variation of the syntax for CLIPS production rules [17] and can be found in Appendix A. In the following sub-subsections we present the syntax and semantics of each one of the three rule types. Furthermore, we present its RuleML compliant syntax.

4.1.1. Deductive Rules

The deductive rule language of R-DEVICE supports reasoning over RDF data represented as objects and derivation of materialized views. The conclusions of deductive rules represent derived classes, i.e. classes whose objects are generated by evaluating these rules over the current set of objects. The derived objects can be maintained incrementally if the user wants. There are two types of rules: truth-maintainable and non truth-maintainable. The truth-maintainable rules preserve truth of derivations: should any of the condition elements that supports a conclusion becomes false, the derived conclusion is deleted, based on a *support list* mechanism explained in section 4.2. The non truth-maintainable rules support conclusions that are no longer true, trading off performance vs. accuracy. Of course, certain applications do not require truth maintenance, but require speed instead.

Furthermore, the rule language supports recursion, stratified negation, path expressions over the objects, and generalized path expressions (i.e. path expressions with an unknown number of intermediate steps). Finally, users can call out to arbitrary built-in or user-defined functions of the host language (CLIPS). However, all the above features supported by the R-DEVICE rule language are compiled away during a pre-compilation phase into a basic first-order logic rule language that supports named attributes (called slots), similarly to F-Logic [29]. The semantics of our rule language is similar to that of function-free Datalog with stratified negation.

The following is an example of an R-DEVICE deductive rule with name `q6` (Appendix B) stating that when there is a `rss:item` resource with a property `rss:title` that contains "RDQL" as a sub-string and a property `rss:link` with value `?link` (a variable), then derive an object of class `result` with a property `link` whose value equals `?link`.

```
(deductiverule q6
  (rss:item (rss:title ?title & :(str-index "RDQL" ?title))) (rss:link ?link)
  =>
  (result (link ?link))
)
```

Although R-DEVICE uses COOL objects, the syntax of rules is as if deductive rules reason about CLIPS templates (i.e. structured facts), because the syntax is simpler. Specifically, each condition element follows the following format:

```
?OID <- (classname (path-expr value-expr) ...)
```

where `?OID` is the (optional) object identifier (or instance name, *not* address) of an object of class `classname`, and `(path-expression value-expression)` are zero, one, or more conditions to be tested on each object that matches this pattern.

When the name of the class is unknown, a variable can be used in place of a concrete class name. For example, in the following rule `doc-title` we seek for the titles of documents created by "*John Smith*".

```
(deductiverule doc-title
  ?x <- (? (dc:title ?t) (dc:creator "John Smith"))
  =>
  (result (document ?t)))
```

However, since the DC namespace is universal, the rule does not require the resources to belong to a certain class, i.e. it ranges over all resources found in the KB. Notice that in this way R-DEVICE can support queries related to properties and not to classes, overcoming the encapsulation of properties inside classes that is caused by the RDF-to-object translation scheme of R-DEVICE (section 3).

Class names can consist of a namespace prefix followed by a colon and a local part name. R-DEVICE allows the use of variables in both the namespace prefix and the local part name. For example, the following condition element applies to instances of classes of the `rss` namespace.

```
(rss:?c (rss:title ?t))
```

A value expression can be a constant or a variable or a constraint or a combination of those, as defined by CLIPS rule syntax. Examples of value expressions are given below.

A path expression is an extension of CLIPS's single ground slot expression. Specifically, in R-DEVICE a path expression can be one of the following:

- A single slot of the class `classname`. For example, the following rule `q5` contains a single condition element (shown in bold) that queries slots `rss:title` and `rss:link` of objects of class `rss:item`:

```
(deductiverule q5
  (rss:item (rss:title ?title) (rss:link ?link))
  =>
  (result (title ?title) (link ?link)))
```

- A single variable denoting *any* slot of class `classname`. For example the following condition element searches for a resource object with an unknown slot whose value is "Smith".

```
(rdfs:Resource (?s "Smith"))
```

- A ground path that consists of a list of multiple slots surrounded by parentheses. The following rule `q7` contains such a path (shown in bold) in the first condition element:

```
(deductiverule q7
  ?x <- (? (dc:title ?tt) (dc:description ?dd) ((etbthes:ETBT dc:subject) ?ss2)
           (dc:identifier ?identifier) ((dcq:RFC1766 dc:language) ?language))
  ?tt <- (? (rdf:value ?t_val) ((dcq:RFC1766 dc:language) ?t_lang))
  ?ss2 <- (? (rdf:value ?subject_val) ((dcq:RFC1766 dc:language) ?subj_lang))
  ?dd <- (? (rdf:value ?desc_val) ((dcq:RFC1766 dc:language) ?desc_lang))
  =>
  (result (title_value ?t_val) (title_language ?t_lang) (subj_val ?subject_val)
          (subj_lang ?subj_lang) (desc_value ?desc_val) (desc_lang ?desc_lang)
          (language ?language)(identifier ?identifier)))
```

  The right-most slot (`dc:language`) should be a slot of the "departing" class. Moving to the left, slots belong to classes that represent the range of the predecessor slots. The value expression in such a pattern (e.g. variable `?language`) actually describes a value of the left-most slot of the path.

- A path that contains one or more single-field variables, i.e. a path whose length is known but some of the steps are not. The above ground path can be turned into such a path:

```
((dcq:RFC1766 ?x) ?language)
```

- A generalized path that contains one or more multi-field variables, i.e. variables that their value is a list. These non-ground paths have an unknown length. The path below can have at least two steps and at most four (given the specific example):

```
((dcq:RFC1766 dc:language $?p) ?x)
```

- A path that contains an encapsulated recursive sub-path, i.e. a sub-path that is traversed an unknown number of times. The following path contains the recursive sub-path `(dcq:references)` which recursively follows resources that reference each other:

```
((dc:title (dcq:references)) ?t)
```

  Recursive paths can be used to express transitive closure queries. For example, the following rule collects all resources (pages) recursively referenced by a certain resource.

```
(deductiverule collect_refs
    (? (uri "http://lpis.csd.auth.gr") ((uri (dcq:references)) ?uri))
```

```
  =>
    (result (uri ?uri))
)
```

Notice that URIs that are reachable following many paths will only be included once in the result and that infinite loops will be avoided, due to the *support list* mechanism (see section 4.2).

Recursive sub-paths can be implicitly included in a path of unknown length. For example in the following generalized path, the multifield variable `$?p` can represent both linear and recursive sub-paths:

```
((dc:title $?p) ?t)
```

Multifield variables can also occur as a value expression, since all RDF properties are treated as multislots. For example, the following pattern retrieves in a list `$?l` all the values for the `rss:link` property of a resource object:

```
(rss:link $?l)
```

On the other hand, if we know that a resource object has many values for one property and we want to iterate over them, the pattern should be:

```
(rss:link $? ?l $?)
```

which means that variable `?l` will eventually become instantiated with all the values of the property `rss:link`. This retrieval pattern is so common that a shortcut is provided which expands to the above pattern during a macro expansion phase.

```
(rss:link ??l)
```

When the value of a specific variable is of no interest then an anonymous variable '`?`' can be used, which is replaced by a singleton system-generated variable during the macro expansion phase.

Selection conditions can be placed inside value expressions, as in CLIPS. For example, the following pattern retrieves the family name in a variable and, at the same time, tests if the slot value does not equal "Smith":

```
(vcard:Family ?last&~"Smith")
```

Conditions can also express disjunction and negation. Only stratified negation is allowed.

The rule conclusion can also contain a set of function calls that calculate the values to be stored at the slots of the derived object. Such calls are placed inside a `calc` construct before the derived class template. For example, the following variation of rule `q2` retrieves the given and family name of a resource object and, using a CLIPS function, concatenates them into a single string that is stored in the slot `full-name` of the derived objects of class `person`.

```
(deductiverule q2-variation
  (? (vcard:Family ?f) (vcard:Given ?v))
  =>
  (calc (bind ?full (str-cat ?v " " ?f)))
  (person (full-name ?full))
)
```

R-DEVICE deductive rules also support aggregate functions and grouping in the form of aggregate attributes whose values are calculated by accumulating and combining attribute values of existing objects. For example, the following rule iterates over all resources and derives one object for each distinct creator, which holds in the `URIs` slot all the resources that he/she has created.

```
(deductiverule ex1-aggregate
  (? (dcq:creator ?c) (uri ?uri))
  =>
  (pages (author ?c) (URIs (list ?uri)))
)
```

Function `list` is an aggregate function that just collects values in a list. There are several other aggregate functions, such as `sum`, `count`, `avg`, etc. Notice that in the above example a grouping is performed because the conclusion contains the slot `author` in addition to the aggregate slot `URIs`. In order to use aggregate functions without grouping, aggregate attribute rules (see section 4.1.3) must be used.

4.1.2. Derived Attribute Rules

Derived attribute rules are rules that derive attributes (for existing objects) whose value is calculated using other attribute values of the same or different object(s). The values for derived attributes are stored and not calculated on-demand. An example of a derived attribute rule is the following:

```
(derivedattrule emp-income
     ?x <- (salesman (salary ?s) (bonus_percentage ?p) (total_sales ?sls))
  =>
     (calc (bind ?total (+ ?s (*?sls ?p))))
     ?x <- (salesman (total_income ?total))
)
```

The above rule states that if the salary, bonus percentage and total sales of a salesman are known then the salesman's total income is calculated by adding to the salary the bonus percentage calculated over the total sales. This rule type is different than deductive rules because here only the value of one attribute of an existing object is affected, whereas in e.g. deductive rule `q2-variation` (above) an entirely new object is derived and the value of one of its attributes is calculated through a function.

The semantics of derived attribute rules are similar to the semantics of a production rule that modifies the attribute of an object, based on the rule condition, which instantiates the identifier of the modified object.

4.1.3. Aggregate Attribute Rules

Aggregate attribute rules are rules that derive attributes (for existing objects) whose value is calculated by accumulating and combining attribute values of multiple other objects. The values for aggregate attributes are also stored, i.e. not calculated on-demand. An example of an aggregate attribute rule is the following:

```
(aggregateattrule ex2-aggregate
     (emp (salary ?s) (department ?d))
  =>
     ?d <- (dept (total_salaries (sum ?s)))
)
```

The above rule states that the total salaries attribute of each department is calculated by summing the salaries of each employee of the department. This rule type is different than deductive rules with aggregate functions because here only the value of one attribute of an existing object is affected, whereas in e.g. deductive rule `ex1-aggregate` (above) an entirely new object is derived and the value of one of its attributes is calculated through aggregation.

The semantics of aggregate attribute rules are similar to the semantics of derived attribute rules. However, here the new value for the modified attribute depends not only on the current variable instantiations of the rule condition, but also on the past collected values. The semantics of aggregate attribute rules seem to violate the Open-World Assumption, because the results are based only on the information found in the (closed) knowledge base of the system. However, rules are non-monotonic since the import of new RDF documents causes results to be re-calculated.

4.1.4. RuleML Syntax of R-DEVICE Rules

The R-DEVICE rule language also has a RuleML [14] compatible syntax. We have tried to keep as close as possible to the latest RuleML version 0.85. However, several features of R-DEVICE could not be captured by the latest RuleML DTDs, so we have developed a new DTD (Figure 11) using the modularization scheme of RuleML, extending the Datalog DTD with the negation-as-failure DTD with OO features.

```
<!ENTITY % CLASSes "NMTOKENS">
<!ATTLIST _rlab
      ruletype (deductiverule | derivedattrule | aggregateattrule) #REQUIRED
      maintainable (yes | no) "yes">
<!ATTLIST var type (single | multi | single-multi) #REQUIRED>
<!ENTITY % recpath.content "(slotname+)">  <!ELEMENT recpath %recpath.content;>
<!ENTITY % genpath.content "(var)">        <!ELEMENT genpath %genpath.content;>
<!ENTITY % slotname.content "(ind|var)"> <!ELEMENT slotname %slotname.content;>
<!ELEMENT _varslot %_slot.content;>
<!ENTITY % _path.content "(slotname|genpath|recpath)+">
<!ELEMENT _path (%_path.content;, %_slot.content;)>
<!ENTITY % rel.content "(ind | var)">
<!ENTITY % _id.content "(ind | var)">      <!ELEMENT _id %_id.content;>
<!ENTITY % atom.content "((_id?,_opr,(_path|_slot|_varslot)*, ...))">
<!ENTITY % _calc.cont "(function_call+)">    <!ELEMENT calc %_calc.cont;>
<!ENTITY % _head.content "(calc?, atom)">
<!ELEMENT aggregate_function_call (var)>
<!ATTLIST aggregate_function_call
        name (sum|count|list|avg|max|min|ord_list|set|string|phrase) #REQUIRED>
<!ELEMENT function_call (%pos_term;)*>
<!ATTLIST function_call name CDATA #REQUIRED>
<!ENTITY % pos_term "(ind | var | function_call)">
<!ENTITY % term "(_not | %pos_term;)">     <!ELEMENT _not (ind | var)>
<!ELEMENT _or (%term;, (%term;)+)>         <!ELEMENT _and (%term;, (%term;)+)>
<!ENTITY % constraint "(_not | _or | _and)">
<!ENTITY % _slot.content "(ind | var | %constraint;|aggregate_function_call)">
<!ENTITY % nafurdatalog_include SYSTEM
                    "http://www.ruleml.org/0.85/dtd/naf/nafurdatalog.dtd">
%nafurdatalog_include;
<!ATTLIST rulebase  xmlns %URI; #IMPLIED  xsi:schemaLocation %URI; #IMPLIED
              xmlns:xsi %URI; #IMPLIED  rdf_import CDATA #IMPLIED
              rdf_export_classes %CLASSes; #IMPLIED rdf_export CDATA #IMPLIED>
```

Figure 11. DTD for the RuleML syntax of the R-DEVICE rule language.

An example of an R-DEVICE rule in RuleML syntax is rule q5 (Appendix B) below:

```
<imp>
  <_rlab ruletype="deductiverule" maintainable="yes">
    <ind>q5</ind>
  </_rlab>
  <_head>
    <atom>
      <_opr><rel><ind>result</ind></rel></_opr>
      <_slot name="link">  <var type="single">link</var>
      </_slot>
    </atom>
  </_head>
  <_body>
    <atom>
      <_opr><rel><ind>rss:item</ind></rel></_opr>
      <_slot name="rss:title">
        <_and>  <var type="single">title</var>
                <function_call name="str-index">
                  <ind>"RDQL"</ind>
                  <var type="single">title</var>
                </function_call>
        </_and>
      </_slot>
      <_slot name="rss:link">  <var type="single">link</var>
      </_slot>
    </atom>
  </_body>
```

```
</imp>
```

There are three types of rules: deductive rules, derived attribute rules and aggregate attribute rules. Classes and objects (facts) can also be declared in R-DEVICE; however, the focus in this paper is the use of RDF data as facts. The input RDF file(s) are declared in the `rdf_import` attribute of the rulebase (root) element of the RuleML document. There are two more attributes in the rulebase element: `rdf_export` declares the address of the RDF file with the results of the rule program to be exported, and `rdf_export_classes` declares the derived classes whose instances will be exported in RDF/XML format.

Further extensions to the RuleML syntax include function calls that are used either as constraints in the rule body or as new value calculators at the rule head. Furthermore, multiple constraints in the rule body can be expressed through the logical operators: `_not`, `_and`, `_or`. Variables belong to three types: single, multi, and a combined form to reflect variable expressions in the previous subsection.

Finally, simple slot expressions have been augmented with the ability to declare path expressions according to the R-DEVICE abilities, i.e. simple ground path expressions, simple path expressions with variables, generalized path expressions, recursive path expressions, etc. Notice that the relation name of the operator can be either a constant (class name) or a variable, since R-DEVICE allows variables to range over class and slot names. Furthermore, each atom element has been augmented with an optional `_id` element to represent the OID of the corresponding resource object.

Notice that despite the quite a few extensions that the R-DEVICE brings to the RuleML syntax, the latter is still valuable for helping interoperation between R-DEVICE and other RuleML-compatible rule systems. Of course, tools are still needed to translate the enhanced rule language of R-DEVICE into more primitive but more widespread rule sublanguages of RuleML and vice-versa.

4.2 Translation of Rules

In this subsection we present how R-DEVICE rules are translated into CLIPS production rules. The semantics of CLIPS production rules [17] 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. In subsequent cycles, new rules may be triggered or un-triggered based on the data modifications. The criteria for selecting rules for the conflict set may be priority-based or heuristically based. Rule condition matching is performed incrementally, through the RETE algorithm.

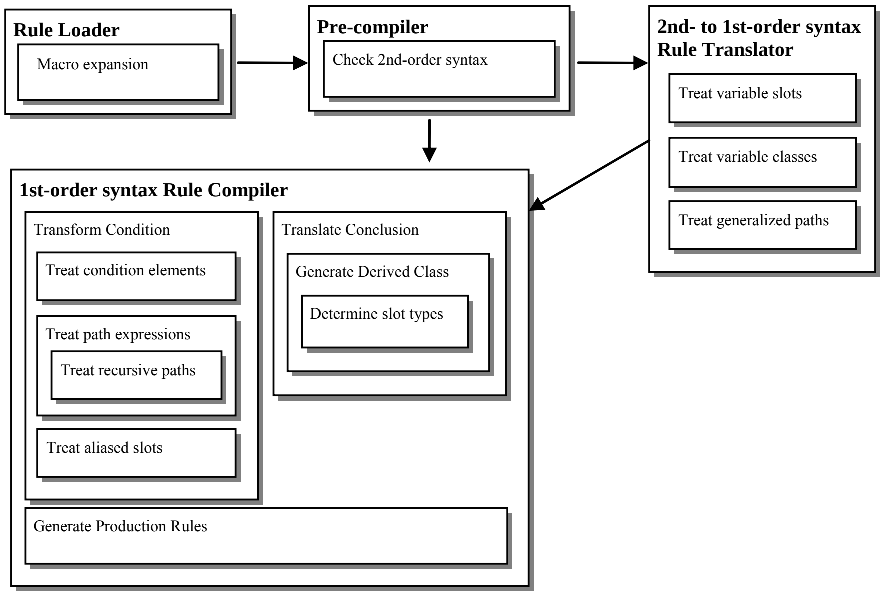


Figure 12. The workflow of the Rule Translator.

Figure 12 shows the major components and sub-components of the rule translator, as well as the workflow of information among them. Initially, the *Rule Loader* reads into CLIPS R-DEVICE rule programs. During loading, some macro expansions, such as the ones mentioned in section 4.1, take place. Then, the *Precompiler* scans rule conditions in order to determine if they have second-order syntax. By second-order syntax, we mean use of variables in the place of class or slot names, or in path expressions. If second-order syntax is present, then the rule is passed through the *Second- to First-order syntax Rule Translator*, which uses the existing schema information to generate a set of deductive rules with first-order syntax that have an equivalent semantics to the second-order syntax rule.

Eventually, rules with first-order syntax, i.e. with no variables in place of classes, slots or paths, are fed to the *First-order syntax Rule Compiler*, which compiles them into production rules. Firstly, the rule condition is translated by turning R-DEVICE syntax for objects into CLIPS syntax. Then, path expressions are transformed into multiple joined condition elements.

In the rest of the section, we discuss the transformation of the rule condition, then we present the translation of all types of first-order R-DEVICE rules and, finally, we describe the transformation of second order rules into sets of first-order rules.

4.2.1. Transformation of the Rule Condition

The condition of R-DEVICE rules is transformed into a condition that follows the CLIPS syntax and the R-DEVICE semantics. More specifically, three types of transformations are performed:

- Path Transformations
- Condition Element Transformations
- Alias Slot Transformations

Path Transformations

Condition elements that contain (ground) path expressions are transformed into multiple condition elements that are chained together using system generated variable names. For example, the following condition element:

```
  ?OID <- (Class (($S_n$ ... $S_i$ ... $S_2$ $S_1$) Val))
```

where `Class` is a class name, S_i are slot names and `Val` is a valid Value and/or constraint expression, is translated into the following set of condition elements:

```
  ?OID <- (Class ($S_1$ ?$var_1$))
  ?$var_1$ <- ($Class_1$ ($S_2$ ?$var_2$))
...
  ?$var_{i-1}$ <- ($Class_{i-1}$ ($S_i$ ?$var_i$))
  ?$var_i$ <- ($Class_i$ ($S_{i+1}$ ?$var_{i+1}$))
...
  ?$var_{n-1}$ <- ($Class_{n-1}$ ($S_n$ Val))
```

where $Class_i$ is the range of the property S_i, and $?var_i$ is a locally unique system generated variable. If property S_i does not have a range, then `rdfs:Resource` is assumed.

When multiple path expressions exist in the same original condition element, then obviously the above transformation occurs for each one of them. However, there is a single `?OID <- (Class ...)` condition element for all path expressions because all paths depart from the same object.

Recursive paths follow a different technique that requires the original rule to be replaced by three other rules with non-recursive paths and an auxiliary derived class. Specifically, assume that the following is e.g. a deductive rule:

```
(deductiverule rule4
     $Condition_{Before}$($Vars_{Before}$)
     ?OID <- (Class (($S_n$ ... $S_{i+1}$ ($RP_n$ ... $RP_1$) $S_{i-1}$ ... $S_2$ $S_1$) Val))
     $Condition_{After}$
  =>
     Conclusion
)
```

where $Condition_{Before}(Vars_{Before})$ is a part of the condition that lies before the condition element that has the recursive path, $Vars_{Before}$ is the set of variables inside $Condition_{Before}$ that are shared with the rest of the rule, (RP_n ... RP_1) is the recursive sub-path of the condition element, $Condition_{After}$ is a part of the condition that lies after the condition element that has the recursive path, and `Conclusion` is the conclusion of the rule.

Rule `rule4` will be replaced by the following set of deductive rules:

```
(deductiverule rule4-1
     $Condition_{Before}$($Vars_{Before}$)
     ?OID <- (Class (($RP_n$ ... $RP_1$ $S_{i-1}$ ... $S_2$ $S_1$) $? ?var1 $?)))
  =>
     (genXX Slots-$Vars_{Before}$ (cnd_obj ?var1))
)

(deductiverule rule4-2
     (genXX Slots-$Vars_{Before}$ (cnd_obj ?var1))
     ?var1 <- (Class-$RP_n$ (($RP_n$ ... $RP_1$) $? ?var2 $?))
  =>
     (genXX Slots-$Vars_{Before}$ (cnd_obj ?var2))
)

(deductiverule rule4-3
     (genXX Slots-$Vars_{Before}$ (($S_n$ ... $S_{i+1}$ cnd_obj) Val))
     $Condition_{After}$
  =>
     Conclusion
)
```

The first rule `rule4-1` navigates the initial part of the path `(`S_{i-1} `...` S_2 S_1`)`, then navigates once the steps of the recursive path `(`RP_n `...` RP_1`)` and finally stores the OIDs of the objects (`?var1`) that lie at the end of this first part of the path into a system-generated derived class `genXX` (into slot `cnd_obj`) along with the values of the variables of the preceding condition that are needed later in the rule. *Slots-Vars*$_{Before}$ are slot expressions that contain the values of those variables; slot names are generated by the system. Actually rule `rule4-1` produces the first layer of objects that lie at the recursive path.

This first layer of objects is used by rule `rule4-2` to recursively navigate all such objects (transitive closure). *`Class-`*RP_n is the range of the last property of the recursive path RP_n. Finally, rule `rule4-3` iterates over all these objects that are reachable through the recursive path (and stored as distinct instances of class `genXX`) and navigates the rest of the path `(`S_n `...` S_{i+1}`)` of the original condition element. The rest of the condition and the conclusion of the original rule are hosted by rule `rule4-3`. These three rules together can replace the original rule in the rule base. Notice that all three rules have only linear ground paths whose transformation has been presented above. In R-DEVICE there is a mechanism that modifies and/or augments the initial rule program with additional rules in order to preserve the semantics of the R-DEVICE rule language.

Condition Element Transformations

After path transformation all condition elements contain only valid CLIPS slot expressions of the form `(`*`slot-name value-expression`*`)`. The next step is to transform R-DEVICE condition element expressions into valid CLIPS condition expressions about COOL objects. This is straightforward, since an expression of the following form:

```
?OID <- (Class slot-expressions*)
```

is transformed into:

```
(object (is-a Class) (name ?OID) slot-expressions*)
```

Condition elements that do not have an "`?OID <-` " expression are also transformed into the pattern above, using a system generated variable, since the OID of each object in the condition is needed for keeping track of the derivators of each derived object (see section 4.2.2).

Alias Slot Transformations

As described in section 3.2, the property hierarchy is treated as slot aliases. When rule conditions contain a super-property in the place of a sub-property special care should be taken. For example, assume that a rule contains the following condition where *`Property`*$_1$ is a super-property of *`Property`*$_2$ and the domain of *`Property`*$_2$ is *`Class`*.

```
(Class (Property1 Val))
```

There are two cases:

- The domain of *`Property`*$_1$ does not include either *`Class`* or any of its direct or indirect super-classes. R-DEVICE replaces *`Property`*$_1$ with *`Property`*$_2$ using the information stored in the `alias` slot of the meta-class of *`Class`*. Otherwise the rule compiler should signal an error since *`Property`*$_1$ is not a direct or inherited slot of *`Class`*.

- The domain of `Property`$_1$ either includes `Class` or some of its direct or indirect super-classes; therefore, `Property`$_1$ is a proper slot of `Class`. In this case one rule for each sub-property of `Property`$_1$ (including `Property`$_1$) is created. In R-DEVICE there is a mechanism that augments the initial rule program with additional rules in order to preserve the RDF(S) semantics.

From the above, it seems that one cannot reason over the subsumption hierarchy of properties except through their domain classes. However, if `Class` in the above example is a variable, then during the second-order to firs-order translation phase (section 4.2.5) it will be replaced by the domain of `Property`$_1$ and one of the above two cases will occur.

4.2.2. Translation of Deductive Rules

The translation of deductive rules depends on whether deductive rules are truth-maintainable or not. The general form of a deductive rule is:

```
(DeductiveRuleType rule1
     Condition
  =>
     (derived-class slot-expressions*)
)
```

where `DeductiveRuleType` is either `ntm-deductiverule` or `deductiverule`, and `derived-class` is the name of the derived class, while `slot-expressions*` are zero or more proper slot-value pairs (see section 4.1.1).

Derived Class Generation

The rule translator has to generate the derived class of the rule conclusion, unless of course it already exists. An important point in generating the derived class is the determination of the slot types. This is based on examining the type of the variables that appear at the rule conclusion. The type of these variables is determined by scanning the condition to find out occurrences of these variables inside slot patterns. Then, the type of the variable can be determined by examining the slot definitions of the corresponding classes. For example, in the following rule, the type of `derived-slot` will be defined by examining the type of the slot `slot`$_1$ of `Class`$_1$ since there exists the shared variable `?var`.

```
(DeductiveRuleType rule2
     (Class₁ (slot₁ ?var))
  =>
     (derived-class (derived-slot ?var))
)
```

If the shared variable appears in two or more different places in the condition, then the minimum common ancestor type will be used. For example, if in the following rule `slot`$_1$ is of type `INTEGER` and `slot`$_2$ is of type `FLOAT`, the type of the `derived-slot` will be `NUMBER`. If both slots are of type `INSTANCE`, then so will be the type of the `derived-slot`. However, using the class reference mechanism of section 3.1 the type of `derived-slot` will be the minimum common ancestor of the classes of both slots in the class hierarchy.

```
(DeductiveRuleType rule2a
     (Class₁ (slot₁ ?var))
     (Class₂ (slot₂ ?var))
  =>
     (derived-class (derived-slot ?var))
)
```

When multiple rules have the same conclusion this means that the final view (derived class) is the union of all the views produced by each rule. Sometimes the conclusions of rules can cause slightly different definitions for the derived class. For example, the following rule `rule3` together with `rule2` above form a pair of rules that have the same derived class as a conclusion.

```
(DeductiveRuleType rule3
     (Class2 (slot2 ?var))
   =>
     (derived-class (derived-slot ?var))
)
```

If *`slot`*$_1$ and *`slot`*$_2$ have the same type, then there is no problem. When the first rule `rule2` is compiled, `derived-class` is generated, and then when later the second rule `rule3` is compiled the existing definition for `derived-class` is used. However, when the types of *`slot`*$_1$ and *`slot`*$_2$ are different, then when `rule3` is compiled the definition for `derived-class` must change. This situation is treated in R-DEVICE by dynamically re-defining the derived class. Specifically, if the existing slot type does not subsume the new type, then the new slot type is appended to the old slot type[2]. The re-definition of the derived class is performed in a way similar to the one described in section 3.3.7.

Non Truth-Maintainable Rules

The translation of non truth-maintainable rules is straightforward and requires a single CLIPS production rule. If the deductive rule `rule1` above is non-truth-maintainable, then the following CLIPS production rule is generated:

```
(defrule rule1-genXX
     (declare (salience (calc-salience derived-class)))
     TransformedCondition
     Check if NewDerivedObject does not exist
   =>
     (make-instance NewDerivedObject of derived-class slot-expressions*)
)
```

`NewDerivedObject` is the OID of the new object of class *`derived-class`* with slot values defined by *`slot-expressions*`*. The OID of derived objects is constructed by concatenating the name of the derived class with the slot values of each derived object. So, for example, if rule `q6` (Appendix B) derives a new object of class result with slot `link` equal to `http://news.com/sports/94224.html`, then the OID of the derived object will be: `result-http://news.com/sports/94224.html`. In this way the existence of a derived object can be efficiently checked through the constructed OID of the derived object. Here we remind that derived objects should be unique regarding the combination of their slot values. *`TransformedCondition`* is the condition that was created by the *Condition Transformation* phase (see section 4.2.1).

The priority of the production rule (called *salience* in CLIPS) is determined by the stratum of the derived class by subtracting the stratum number from a fixed salience (2000). The stratum is an integer that is used by semi-naive evaluation of logic rule programs with stratified negation [40] to indicate the order by which the rules (based on their conclusion) are evaluated. Rules that derive classes with low stratum are evaluated before rules that derive classes with higher stratum, thus they must have a higher salience in CLIPS. Figure 13 shows the

[2] In COOL a slot can have multiple types with disjunctive semantics.

production rule generated by R-DEVICE for the deductive rule `q6` (Appendix B), in its non-truth-maintainable version.

```
(defrule q6-gen54
   (declare (salience (calc-salience result)))
   (object (name ?gen53)  (is-a rss:item)
           (rss:title ?title&:(str-index "RDQL" ?title))  (rss:link ?link))
   (test (not (instance-existp (symbol-to-instance-name (sym-cat result ?link)))))
   =>
   (bind ?oid (symbol-to-instance-name (sym-cat result ?link)))
   (make-instance ?oid of result  (link ?link))
)
```

Figure 13. CLIPS production rule generated for the non-truth-maintainable deductive rule q6 (Appendix B).

For each translated R-DEVICE rule an object is generated that keeps meta-information, such as the identifier(s) of the generated production rule(s), the names of the classes in their condition, and the name of the derived class. Information about the derived class, such as the stratum of the derived class and the list of the identifiers of the deductive rules that have this class as their conclusion, are kept separately in a meta-class.

Truth-Maintainable Rules

The translation of truth-maintainable rules is more difficult since it requires two CLIPS production rules in order to maintain the materialized derived views: one for creating the derived object when the condition is true, and one for deleting the derived object when the condition becomes false. If the deductive rule `rule1` above is truth-maintainable, then the following pair of CLIPS production rules is generated:

```
(defrule rule1-genXX
     (declare (salience (calc-salience derived-class)))
     TransformedCondition
     Check    if NewDerivedObject does not exist
              Or if it exists and the couple rule1-genXX-A (A is the current set of derivators) is not a member
                                                                                           of its support list
   =>
     (if NewDerivedObject exists
       then
          Add the couple rule1-genXX-A to its support list
       else
          (make-instance NewDerivedObject of derived-class slot-expressions*)
          Add the couple rule1-genXX-A to the support list of NewDerivedObject
     )
)
(defrule rule1-genYY
     (declare (salience 2000))
     Check if NewDerivedObject exists and for each couple rule1-genXX-A (A is a set of derivators) in its support list
          Check if TransformedCondition (instantiated with objects in A) is false
   =>
     Remove the couple rule1-genXX-A from the support list of  NewDerivedObject
     (if  the new support list is empty
       then  delete  NewDerivedObject
     )
)
```

The "positive" production rule `rule1-genXX` creates a new derived object when the condition *`Transformed-Condition`* is true. Furthermore, the rule checks if the *`NewDerivedObject`* does not yet exist, similarly to the non-truth-maintainable deductive rule case. However, here there is a difference because derived objects should be correctly maintained during updates of the base data. Each derived object maintains a *support list*, which contains all the *derivators* of the derived object. A *derivator DV* of a derived object *DO* is a couple *DR-CO* where *DR* is a deductive rule which derives *DO*, when its positive condition elements are instantiated by the set of objects *CO*. Derivators are needed for maintaining the derived object, because the same object can be derived from many different rules and from many different objects, since the rule conclusion is a projection of the values of the variables of the condition elements which in turn are a projection of the objects that they get instantiated with.

The positive production rule checks if the derived object already exists. If not, its action generates the object and adds the current derivator to the object's support list. If the object already exists, the rule checks if the current derivator already exists in its support list. If it does, then it means that the inference procedure is about to enter an infinite loop, since the same objects derive the same conclusion; therefore, the production rule just does not fire. If the current derivator does not exist in the derived object's support list, the production rule action adds it.

The "negative" production rule `rule1-genYY` checks if the derived object already exists and for each derivator *A* in the support list that was produced by the positive production rule `rule1-genXX` it checks if the condition *`TransformedCondition`* is false, when it is instantiated with objects from set *A*. If such a derivator is found, then it means that it does no longer support the conclusion that the derived object stands for and it must be deleted from its support list. If this was the last derivator and the support list is now empty, it means that the derived object is no more concluded by any rule-objects combination, so it is deleted.

Execution of truth-maintainable rules is more expensive than non-truth-maintainable rules since negative production rules monitor at every production cycle if the derivators of all derived objects still satisfy the condition of the deductive rule. Since the deductive rule language supports negation-as-failure, rule conclusions can be invalidated both by object deletions and insertions; therefore, monitoring should occur both on deletions and insertions. The salience of the positive production rule is calculated in the same way as with non-truth-maintainable rules. On the other hand, “negative” production rules have a fixed salience 2000, i.e. when multiple updates exist first the old view is maintained and then the new view is calculated.

```
(defrule q6-gen54
   (declare (salience (calc-salience result)))
   (object (name ?gen53)  (is-a rss:item)
           (rss:title ?title&:(str-index "RDQL" ?title))  (rss:link ?link))
   (not (object (name ?DO&:(eq ?DO (symbol-to-instance-name (sym-cat result ?link))))
                (is-a result)  (link ?link)  (derivators $? +++ ? ?gen53 +++ $?)))
   =>
   (bind ?oid (symbol-to-instance-name (sym-cat result ?link)))
   (if (instance-existp ?oid)
      then   (slot-insert$ ?oid derivators 1 +++ q6-gen54 ?gen53 +++)
      else   (make-instance ?oid of result
                     (link ?link)  (derivators +++ q6-gen54 ?gen53 +++)))
)

(defrule q6-gen55
   (declare (salience 2000))
   (object (name ?derived-object)  (is-a result)
           (link ?link)  (derivators $?DER-B +++ q6-gen54 ?gen53 +++ $?DER-A))
   (or  (test (not (all-instance-existp (create$ ?gen53))))
        (and (object (name ?gen53)  (is-a rss:item))
             (not (object (name ?gen53)  (is-a rss:item)
                          (rss:title ?title&:(str-index "RDQL" ?title))  (rss:link ?link)))))
   =>
   (if (= (length$ (create$ $?DER-B $?DER-A)) 0)
      then   (send ?derived-object delete)
      else   (message-modify-instance ?derived-object  (derivators $?DER-B $?DER-A)))
)
```
Figure 14. CLIPS production rules generated for the truth-maintainable deductive rule q6 (Appendix B).

Figure 14 shows the pair of production rules generated by R-DEVICE for the deductive rule `q6` (Appendix B). Actual production rules differ slightly from the abstract production rules presented above due to some low-level implementation techniques that are specific to CLIPS, which are beyond the scope of this paper.

Notice that truth maintenance can be turned on or off, even for truth-maintainable rules, according to user preference. In this way, users can just bulk-load the knowledge base without having the expensive "negative" rules constantly monitoring the knowledge base. Users can just turn on truth-maintenance at any time and check the consistency of the knowledge base (user-defined consistency checkpoint).

4.2.3. Translation of Derived Attribute Rules

The translation of derived attribute rules requires two CLIPS production rules in order to maintain the derived attribute value: one for inserting the value of the derived attribute when the condition is true, and one for deleting the value when the condition becomes false. The difference between truth-maintainable and non-truth-maintainable rules is only in the "negative" production rule, which is just not created in the latter case.

The general form of a derived attribute rule is:

```
(derivedattrule rule2
     Condition
   =>
     CalculationExpressions
     ?OID <- (Class slot-expressions*)
)
```

The following pair of CLIPS production rules is generated:

```
(defrule rule2-genXX
     (declare (salience 1000)))
     TransformedCondition
     (not ?OID <- (Class calc-slot-expressions*))
   =>
     CalculationExpressions
     (modify-instance ?OID slot-expressions*)
)
(defrule rule2-genYY
     (declare (salience 500)))
     ?OID <- (Class calc-slot-expressions*)
     (not TransformedCondition)
   =>
     (modify-instance ?OID null-slot-expressions*)
)
```

`calc-slot-expressions` are the slot expressions of the conclusion pattern where the calculation expressions *`CalculationExpressions`* have been incorporated in the form of functional constraints. *`null-slot-expressions`* are the slot expressions of the conclusion pattern where all the derived attribute values have been set to null.

The positive production rule `rule2-genXX` checks if the condition is true and if the object `?OID` with the derived attribute already has the derived values that the rule is about to insert. In this case, the rule simply does not fire. If the condition is true, the derived attributes of the object `?OID` gets the values calculated by the condition and the functions contained within *`CalculationExpressions`*.

The negative production rule `rule2-genYY` checks whether the object `?OID` exists and has the values inserted by the positive rule. Furthermore, it checks if the condition of the derived attribute rule does not hold any more, which means that the derived attribute values should no longer exist. If the above conditions occur then all the derived attributes of the rule get a null value.

The salience of derived attribute rules is lower than deductive rules. This allows for combined derivations: deductive rules can derive objects, while derived attribute rules can further derive attributes of the generated objects using more complex calculations. The salience of positive production rules is 1000 while the salience of negative production rules is 500.

4.2.4. *Translation of Aggregate Attribute Rules*

The translation of aggregate attribute rules requires two CLIPS production rules in order to maintain the aggregate attribute value: one for inserting a new value into the collection of values that will calculate the aggregate attribute when the condition is true, and one for deleting an existing value from the above collection when the condition becomes false. The difference between truth-maintainable and non-truth-maintainable rules is only in the "negative" production rule, which is just not created in the latter case.

The general form of an aggregate attribute rule is:

```
(aggregateattrule rule3
     Condition
   =>
     CalculationExpressions
     ?OID <- (Class (Slot (Aggregate-function ?var)))
)
```

where `?var` is a variable that occurs inside *`Condition`*, and *`Aggregate-function`* is one of the aggregate functions that R-DEVICE supports or a user-defined function.

The calculation and maintenance of aggregate attributes is achieved through an auxiliary object that is unique for each object that hosts an aggregate attribute and for each different aggregate attribute, if multiple such attributes exist for the same object. This auxiliary object is an instance of the class of the aggregate function. There is a different class for each aggregate function (Table 2), in order to model the different algorithm that each function requires to calculate the corresponding aggregate value.

The structure of auxiliary objects includes:

- A slot `instance` that stores the OID of the object that hosts the aggregate attribute.
- A slot `attribute` that stores the name of the aggregate attribute.
- A multi-slot `values` that stores all the values that are collected to calculate the value of the aggregate attribute, through the aggregate function.
- A multi-slot `objects` that stores the OIDs of the objects of the positive condition elements of the rule, similarly to the *derivators* in deductive rules in section 4.2.2. This is needed in order to correctly calculate and maintain the aggregate attribute value.

```
(defclass avg
   (is-a aggregate-function)
)

(defmessage-handler avg calc-result ($?result)
   (if (> (length$ $?result) 0)
     then    (/ (sum$ $?result) (length$ $?result))
     else    0
   )
)
```

Figure 15. Method `calc-result` for aggregate function `avg`.

Each aggregate function class has an (overloaded) method called `calc-result`, which actually calculates the value of the aggregate attribute based on the values collected in the `values` slot. Figure 15 shows an example of the code for this method for the `avg` function; `$?result` is the list of values to be averaged by dividing their sum by their length (count). The set of aggregate functions that R-DEVICE supports (along with their meaning) is shown in Table 2. Users can also define their own aggregate functions by providing a new class for each new aggregate function, as a subclass of class `aggregate-function` (see Figure 15) and a method `calc-result` for calculating the aggregate function value by combining the collected values.

Table 2. Aggregate functions of R-DEVICE.

Name	Meaning
`sum`	The sum of numerical values
`count`	The cardinality of values
`avg`	The average of numerical values
`max`	The maximum among numerical or symbolic values
`min`	The minimum among numerical or symbolic values
`list`	Sequence of values (just returns slot `values`)
`ord_list`	Sorted sequence of numerical or symbolic values
`set`	Set of values (no duplicates)
`string`	Concatenation of symbolic values
`phrase`	Concatenation of symbolic values with a white space between them

For each aggregate attribute rule, the following pair of CLIPS production rules is generated:

```
(defrule rule3-genXX
     (declare (salience 1000)))
     TransformedCondition
     Check    If the corresponding auxiliary object AO does not exist:
                      (not AO <- (Aggregate-function (instance ?OID) (attribute Slot)))
              Or if it exists, it does not contain the value ?var coming from the set A of current derivators
  =>
     CalculationExpressions
     If the corresponding auxiliary object AO does not exist, then create it
     Add ?var to slot values of AO
     Add set A to slot objects of AO
     Store the result of the method calc-result applied over the new value of the slot values
          to the attribute Slot of the object ?OID
)
(defrule rule3-genYY
     (declare (salience 500)))
     Check    if the auxiliary object AO exists:
                            AO <- (Aggregate-function (instance ?OID) (attribute Slot))
                For each value V in the slot values and the corresponding set A of derivators in the slot objects
                    Check if TransformedConditionCalc (instantiated with objects in A) is false
  =>
     Remove V from slot values of AO
     Remove set A from slot objects of AO
     Store the result of the method calc-result applied over the new value of the slot values
          to the attribute Slot of the object ?OID
)
```

`TransformedConditionCalc` is the transformed condition of the original rule (according to the scheme of section 4.2.1) where the calculation expressions *`CalculationExpressions`* have been incorporated in the form of functional constraints.

The positive production rule `rule3-genXX` checks if the condition is true and if the auxiliary object that corresponds to the object of the conclusion and the specific aggregate attribute does not exist. However, even if the latter exists, the rule can still fire if the value `?var` does not exist in the slot `values` of the auxiliary object or the current set of derivators does not occur in the slot `objects`. If the condition is true, the auxiliary object is either created or just retrieved, the slots `values` and `objects` are augmented with the corresponding values, and method `calc-result` is invoked to calculate the value of the aggregate function over the new set of collected values. The latter is finally stored in the corresponding attribute of the object `?OID`.

The negative production rule `rule3-genYY` checks if the auxiliary object that corresponds to the object of the conclusion and the specific aggregate attribute exists. Then, for each value *V* in the slot of collected values (and for its corresponding set of derivators *A*) it checks whether the condition ***`TransformedCondition`*** is false when it is instantiated with objects from set *A*. Notice that the condition is augmented with ***`CalculationEx-pressions`*** in the form of functional constraints. If such a value-derivators combination is found, then it means that the derivators *A* no longer support the existence of value *V* and the latter must be deleted from the collected values, along with the corresponding derivators. Furthermore, the `calc-result` method is called again to calculate the new value for the aggregate function that is stored at the corresponding attribute of the object `?OID`.

The saliencies of aggregate attribute rules are exactly the same as derived attribute rules. Notice that although the saliencies of derived and aggregate attribute rules are fixed, R-DEVICE provides an extensibility hook, so that future rule applications built on top of R-DEVICE can implement their own rule priority mechanism. In fact, this has already been exploited to implement defeasible logic rules on top of R-DEVICE [3].

4.2.5. Translation of Second-Order Syntax into First-Order Syntax

In this subsection we present how R-DEVICE translates rule with second-order syntax, i.e. rules that contain variables in place of class names and/or slot names, into sets of first-order rules using the RDF schema. One of the main concerns of this step is to produce as few deductive rules as possible, for efficiency reasons. There are three types of second-order rule syntaxes that are handled in different ways:

- Variable class names
- Variable slot names
- Generalized paths (i.e. paths with a variable step)

Using the loaded RDF Schema at compile-time implies that the second-order translation process is only meaningful under a Closed-World Assumption. This is so because the grounding of variables during the translation considers only the classes and properties loaded at compile-time and not any class that could possible be loaded later. However, it is not very difficult to extend the current compilation scheme of R-DEVICE into incremental run-time compilation, because the rule compilation phase is implemented as a set of production rules that could be possibly triggered at run-time, after each time a new RDF Schema document is imported.

Variable Class Names

When the class name is a variable, then R-DEVICE should generate as many deductive rules as the number of existing classes. Since this is extremely naïve, R-DEVICE selects only those classes that have a set of slots (including inherited ones) that is a superset of the slots that appear inside the corresponding condition element. For example, rule `q1a` (Appendix B) has an anonymous variable in the place of a class name:

```
(deductiverule q1a
  ?x <- (? (email:message-id '123456@example.com'))
  =>
  (result (email ?x)))
```

There will be generated as many rules as many classes exist in the schema that have `email:message-id` as a slot. Notice that if there is no class that satisfies the condition above, then no deductive rule is generated and the second-order rule does not affect the rule base. Notice also that when the above algorithm produces many classes that belong to the same inheritance path in the class hierarchy, R-DEVICE keeps only the most general class(es) from this set. However, this optimization is not used when the variable that represents the name of the class is not anonymous and is used later in the rule, since its value is important for the rule.

Another case is when the condition element that has the variable class name has only one slot pattern whose slot expression is also a variable, as in the following rule `q3` (Appendix B):

```
(deductiverule q3
  data:x <- (? (?property ?value))
  ?property <- (rdf:Property (rdfs:range $? ?t $?))
  =>
  (result (property ?property) (value ?value) (type ?t)))
```

In this case the above class filtering optimization cannot be performed. The only way to determine the class of this condition element is to check the object identifier expression. If it is a variable, then the condition is checked for another occurrence of the variable, so the type of the variable can be discovered by examining the type of the slot where it is referenced. If it is not a variable, then its type can only be discovered if the object already exists. If none of the above cases occurs, then the only choice for R-DEVICE is to generate as many rules as the classes. The class hierarchy optimization that was mentioned above is applied.

The cases where the class name is a combination of a namespace and a local name, and either or both of them are variables are treated in a similar manner. Initially, all the namespaces and/or all the class of the same namespace are iterated in order to produce full concrete class names. Then, the algorithm presented above is executed.

Variable Slot Names

When a slot name in the rule condition is a (single) variable and the class name of the corresponding condition element is not, the set of all the slot names of the class is retrieved and the variable slot can take values from this set, excluding those that already appear in the condition element. R-DEVICE generates as many deductive rules as the number of instantiations the variable slot can take.

For example, consider the following condition element:

```
?OID <- (Class (Slot1 Val1) (?VarSlot Val2))
```

The variable `?VarSlot` can be instantiated from all the slot names of ***`Class`*** (including inherited ones) except ***`Slot1`***. Another optimization that is performed is to also exclude those slot names whose data type is not compatible with the type of value ***`Val2`***, if the latter can be determined. If ***`Val2`*** is a constant then its type can be easily determined. If it is a variable, then the condition is checked for another occurrence of the variable, so the type of the variable can be discovered by examining the type of the slot where it is referenced.

A different case is when the variable slot is a multi-variable, as in the following example:

```
?OID <- (Class (Slot1 Val1) ($?VarSlot Val2))
```

In this case only one rule is generated by augmenting the above condition element with as many (*SlotName Val2*) expressions as the number of valid instantiations the variable slot *?VarSlot* can take, using the optimizations discussed above.

Generalized Paths

Path expressions can contain either a single-valued or a multi-valued variable whose meaning and treatment is quite different. A single-valued variable in a path expression means that a single step in the path is unknown. This case is quite similar to the combination of variable class and variable slot that was discussed above. Specifically, assume the following condition element:

```
  ?OID <- (Class ((Sn ... Si+1 ?VarSlot Si-1 ... S2 S1) Val))
```

which actually corresponds to the following set of condition elements (see *Path Transformations* in section 4.2.1):

```
  ?OID <- (Class (S1 ?var1))
  ?var1 <- (Class1 (S2 ?var2))
...
  ?vari-1 <- (Classi-1 (?VarSlot ?vari))
  ?vari <- (?VarClass (Si+1 ?vari+1))
...
  ?varn-1 <- (Classn-1 (Sn Val))
```

where ?VarClass is the unknown range of the unknown property ?VarSlot. Using a combination of the algorithms presented above the alternative values of slot ?VarSlot can be discovered with greater accuracy than a simple variable slot, because ?VarSlot is more constrained by participating in a path where the preceding and following steps are already known. Specifically, the range of ?VarSlot must be class that hosts the concrete slot S_{i+1} and its domain must be the concrete class $Class_{i-1}$. All the allowed instantiations of ?VarSlot lead to the generation of different deductive rules.

A multi-valued variable in a path expression means that there is an unknown sub-path of the path whose length is also unknown. For example, consider the condition element below:

```
  ?OID <- (Class ((Sn ... Si+1 $?VarSlot Si-1 ... S2 S1) Val))
```

In this case R-DEVICE performs a graph search in the graph formed by classes and slots, through object type properties, having as a starting position the class that is the range of slot S_{i-1}. The set of ending positions of the search includes all the classes that are the domains (even through inheritance) of the slot S_{i+1}. This graph search may end up with multiple alternative solution paths, including path with zero length. For each of these solution paths a different deductive rule is generated. Notice that in case no solution to the graph search problem is found, then no deductive rule will be generated. Furthermore, the case of a single-valued attribute (presented above) can be considered a special case of a multi-valued attribute: the graph search is constrained to find solution paths of length one.

Generalized paths can also contain recursive sub-paths. In this case, the translation is a combination of the technique (graph search) we presented above and the transformation of recursive paths that has been presented in section 4.2.1.

4.3 Extracting Rule Conclusions as RDF Documents

The deductive rule language materializes the conclusions of rules as concrete objects. Since we provide a translation of RDF statements into objects, the inverse is also possible. After production rules have been executed and all the derived objects have been generated, the *RDF Extractor* generates an RDF/XML document and returns it to the user through a Web server, as the result of the user's program. The document contains both RDF definitions for the schema of the desired derived classes and, of course, for the instances of the derived classes. Notice that the user can specify the name of the extracted file as well as the names of the derived classes that the file will contain (along with their instances, of course).

Initially the result document contains namespace definitions for rdf/rdfs and for the exported document. Then the derived classes are defined as `rdfs:Class` elements, followed by `rdfs:Property` elements for each of their slots that was defined at the deductive rule conclusion. Domains and ranges of the properties are obtained from the COOL definition of each derived class. The domain for all the properties of a class is the class itself, while the range can be one of the following:

- If the slot is of type `SYMBOL` and/or `STRING`, the property range is `rdfs:Literal`.
- If the slot is of type `INTEGER`, the property range is `xsd:integer`.
- If the slot is of type `FLOAT`, the property range is `xsd:float`.
- If the slot is of type `INSTANCE`, the property range is a class whose name is obtained by the `class-refs` slot of the meta-class (see section 3.2). When the referenced class is a system-generated class (e.g. in the case of multi-range properties), then multiple property ranges are generated for each one of the superclasses of the system-generated class.
- If the slot has any other combination of types, then there is no range constraint for the property.

Notice that in the fourth case the RDF schema for the referenced class(es) must be included in the result document, unless it is not a derived but a base RDF class, i.e. an RDF class whose definition has been imported from a namespace. In this case, the document header is enriched with the namespace address and no further schema definition is included. Otherwise, definitions for the referenced class and its properties are included in the RDF result document. The same actions are recursively repeated for all classes that are reachable by reference slots from the initial class.

Below all class and property definitions, appear RDF statements about the instances of those classes. For each of the initial derived classes, all its objects are included using class names as outside elements and property names as inside elements. Only slots/properties that do have a value are included. Properties with multiple values are represented as multiple consecutive elements. Finally, objects recursively referenced from the above objects are also included in the result document. When instances of base RDF classes are included the outside element is `rdf:Description`, because type information is included as one or more `rdf:type` properties. The URIs of the derived objects are constructed from the URI of the R-DEVICE system[3], the name of the exported file and a uniquely generated anchor ID, constructed from the class name and consecutive integers. The URI of instances of base classes is taken from the `uri` slot of each object.

[3] `http://startrek.csd.auth.gr/r-device/export/`

Below we present an R-DEVICE rule `example` over the RDF document of Figure 8, which retrieves the title of an ODP topic that has at least one associated page, along with the titles of all associated pages. The class definition for the derived class `result` and its single instance are shown in Figure 16. Finally, Figure 17 shows the exported results in RDF format.

```
(deductiverule example
     (dmoz:Topic (dc:title ?t) (dmoz:link $? ?l $?))
     ?l <- (dmoz:ExternalPage (dc:title ?lt))
  =>
     (result (title ?t) (link_title ?lt))
)
```

```
 (defclass result
   (is-a DERIVED-CLASS)
   (slot title (type ?VARIABLE))
   (slot link_title (type ?VARIABLE))
   ...
)

([resultArtsJohn phillips Blown glass] of result
   (source rdf)
   (derivators +++ gen2 [gen33] [gen34] +++)
   (title "Arts")
   (link_title "John phillips Blown glass")
)
```

Figure 16. Derived class definition and derived object for R DEVICE rule `example`.

```
<!DOCTYPE rdf:RDF [
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
    <!ENTITY r_device "http://startrek.csd.auth.gr/r-device/export/example-result.rdf#">]>

<rdf:RDF  xmlns:rdf='&rdf;'  xmlns:rdfs='&rdfs;'  xmlns:r_device='&r_device;'>
  <rdfs:Class rdf:about='&r_device;result'> </rdfs:Class>
  <rdf:Property rdf:about='&r_device;title'>
       <rdfs:domain rdf:resource='&r_device;result'/>
  </rdf:Property>
  <rdf:Property rdf:about='&r_device;link_title'>
       <rdfs:domain rdf:resource='&r_device;result'/>
  </rdf:Property>
  <r_device:result rdf:about="&r_device;result1">
       <r_device:title>Arts</r_device:title>
       <r_device:link_title>John phillips Blown glass</r_device:link_title>
  </r_device:result>
</rdf:RDF>
```

Figure 17. Exported results for R DEVICE rule `example`.

5 Performance Results

In this section we present some performance tests we have conducted for R-DEVICE, comparing our object-oriented RDF model with the "traditional" triple-based RDF model. Notice that for the sake of the comparison we have implemented a very simple triple-based model on R-DEVICE. Specifically, each RDF triple is directly mapped to a single COOL object, as an instance of the `rdf-triple` class, with three slots: `subject`, `predicate`, and `object`.

The RDF data we used for our experiments are taken from the Open Directory Project (ODP)[4]. More specifically, we have used fragments from the ODP structure and content files, conducting 3 sets of tests with files containing 1K, 10K, and 100K triples. These tests are not complete and do not claim to cover the entire spectrum of possible tests one could perform on reasoning over RDF data. Our intention in this paper is to demonstrate that our OO approach leads to increased inference performance compared to the triple-based approach and, as results indicate, the ODP data set suffices for this. In the future we will test the performance of R-DEVICE with other RDF files, using more inference cases and more complex schemas, such as the Lehigh University Benchmark [25].

The experiments were performed on an Intel Pentium IV 2.4GHz PC with 512MB main memory and Windows XP Professional. As mentioned in section 2, loading/translating (collectively called *importing*) of RDF triples can be performed either in a single step or in a streaming fashion. In order to estimate which is the optimal number of triples to be loaded/translated per cycle of the iteration we have performed RDF triple import in a single step for various number of triples and we have calculated the average import time per triple (Table 3). Results show that the average import time per triple is optimal for 10K triples; therefore, we consider this to be the optimal number of triples per iteration for the streaming triple import.

Table 3. Single-step triple import time.

No. of triples	Total (sec)	Avg / triple (msec)
1 K	2.198	2.198
10 K	10.220	1.022
100 K	215.879	2.159
1000 K	23821.446	23.821

Next, we performed RDF triple import in a streaming fashion with 10K as a triple limit. Table 4 shows the total import time and average import time per triple, as well as the number of cycles the import algorithm performed. As it can be seen from Figure 18 and Figure 19 the total and average per triple import times are significantly improved in the streaming case, compared to the single-step case.

Table 4. Streaming triple import time.

No. of triples	Cycles	Total (sec)	Avg / triple (msec)
1 K	1	2.198	2.198
10 K	1	10.220	1.022
100 K	10	77.912	0.779
1000 K	100	1312.033	1.312

Notice that the triple import times reported above are for ODP RDF documents that do not have any predefined RDF Schema. The actual schema for the ODP data is "discovered" at run-time by the RDF triple translator by applying the RDF semantics. When an RDF Schema for the ODP documents is provided, import time is slightly improved, especially for few triples, as Table 5 and Figure 20 show. However, since there are not great differences between times in Table 4 and Table 5 we can conclude that schema re-definition does not incur a very high overhead as a fraction of total triple import time.

Table 5. Streaming triple import time (with schema).

No. of triples	Total (sec)	Avg / triple (msec)
1 K	0.989	0.989
10 K	6.758	0.676
100 K	62.527	0.625
1000 K	1057.747	1.058

[4] `http://dmoz.org`

Tables 6-8 summarize the results from running the RDF (non-truth-maintainable) rule cases of Appendix C, for both the object-oriented and the triple-based approaches. Rule cases include chaining of objects (cases 4-10), negation (case 9), and recursion (case 10). The performance of the equivalent truth-maintainable rules is on average 10 times slower mainly because of the existence of the "negative" production rules which check at every cycle for all the derived objects if their existence is still supported. In Table 9 we have included the rule compilation times for each rule case (for both approaches) for comparison with the actual rule execution time. These times get to practically zero when rules are already compiled and loaded just for execution. The second column of the tables 6-8 include the number of objects that were derived by running each rule case, while the last column shows the speed-up of the OO-RDF model compared to the triple-based RDF model (i.e. inferencing time of the triple-based RDF model divided by the inferencing time of the OO-RDF model).

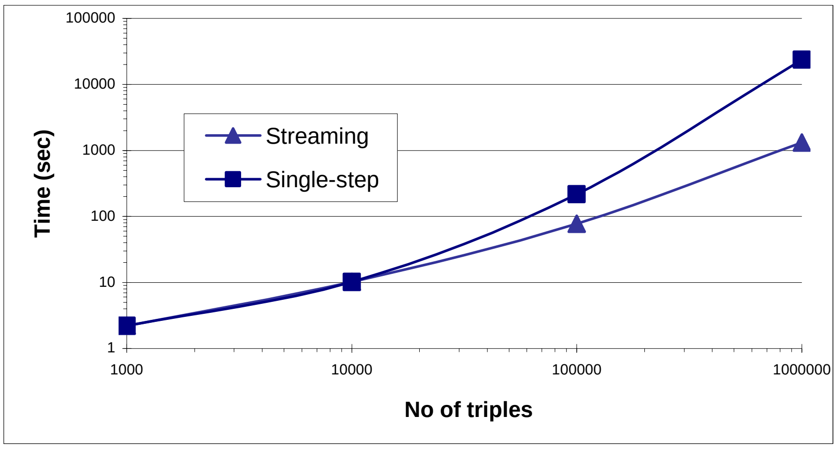


Figure 18. Triple import time (single-step vs. streaming).

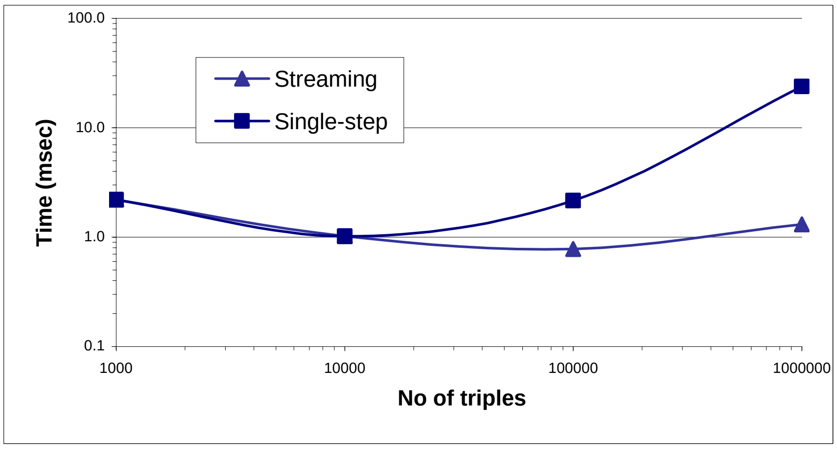


Figure 19. Average import time per triple (single-step vs. streaming).

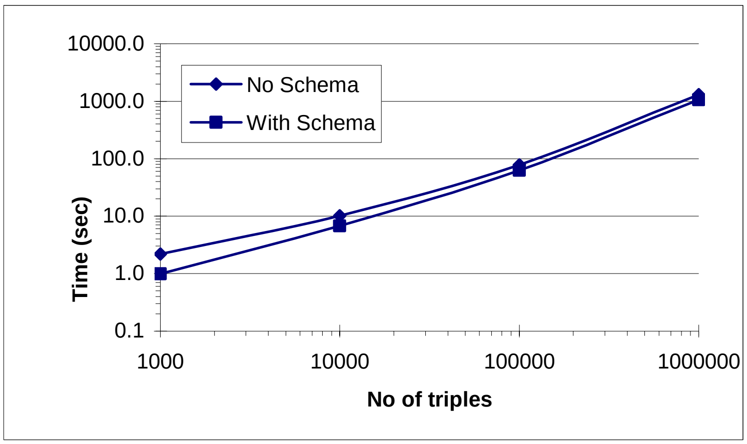


Figure 20. Triple import time (with schema vs. no schema).

Results clearly show that for almost all rule cases inferencing for the OO-RDF model is significantly faster, except rule case 0, where only one triple is included in the condition. Improvement goes up to almost 5 orders of magnitude in rule case 3, for 100K triples! Figures 21-31 show how both approaches scale-up to the number of triples (in log-log axes). It is clear that the object-oriented approach scales-up almost linearly in all cases. Furthermore, the OO approach scales-up better than or at worst similarly with the triple-based approach. Finally, Figures 32-42 show the speed-up scaling curves, again in log-log axes. Except for rule case 0 (Figure 32), where speed-up is almost 1, in all other cases speed-up increases monotonically with the number of triples.

Table 6. Performance Results for various Rule Cases (1K triples).

Rule Case	Derived Objects	Triple RDF model (sec)	OO RDF model (sec)	Speed-up
0	205	0.007253	0.00604	1.20
1	16	0.001813	0.00070	2.58
2	3	0.001429	0.00030	4.81
3	1	0.001648	0.00021	7.69
4	2	0.001593	0.00052	3.09
5	2	0.002033	0.00054	3.74
6	12	0.005275	0.00247	2.13
7	162	0.031319	0.03643	0.86
8	12	0.005769	0.00885	0.65
9	0	0.005879	0.00076	7.70
10	168	0.033516	0.04060	0.83

Table 7. Performance Results for various Rule Cases (10K triples).

Rule Case	Derived Objects	Triple RDF model (sec)	OO RDF model (sec)	Speed-up
0	1997	0.137363	0.11538	1.19
1	269	0.873626	0.01703	51.29
2	30	0.857143	0.00231	371.43
3	1	0.840659	0.00034	2508.20
4	2	0.862637	0.00500	172.53
5	2	0.978022	0.00527	185.42
6	194	2.582418	0.28571	9.04
7	975	7.307692	2.80220	2.61
8	80	2.362637	0.63187	3.74
9	0	9.725275	0.08077	120.41
10	358	2.802198	0.86264	3.25

Table 8. Performance Results for various Rule Cases (100K triples).

Rule Case	Derived Objects	Triple RDF model (sec)	OO RDF model (sec)	Speed-up
0	17697	15.23626	12.41758	1.23
1	1014	88.33516	0.35714	247.34
2	96	88.49451	0.02253	3928.29
3	1	87.93407	0.00101	86978.26
4	2	90.11538	0.03791	2376.96
5	2	101.2527	0.04176	2424.74
6	1000	218.7692	15.98901	13.68
7	11775	891.2637	268.57143	3.32
8	825	259.2857	42.58242	6.09
9	0	26189.84	10.71429	2444.38
10	361	109.6154	7.41758	14.78

Table 9. Rule Compilation Times

Rule Case	OO RDF model (sec)	Triple RDF model (sec)
0	0.027	0.011
1	0.016	0.016
2	0.022	0.016
3	0.038	0.016
4	0.027	0.038
5	0.055	0.038
6	0.071	0.044
7	0.033	0.044
8	0.038	0.049
9	0.044	0.060
10	0.077	0.077

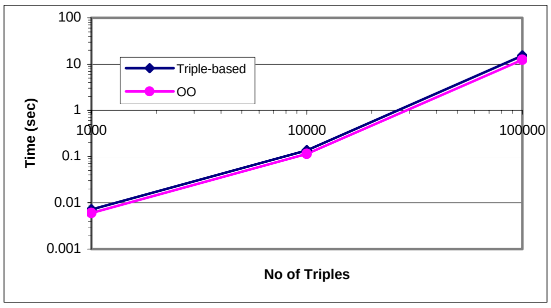


Figure 21. Performance results for Rule Case 0.

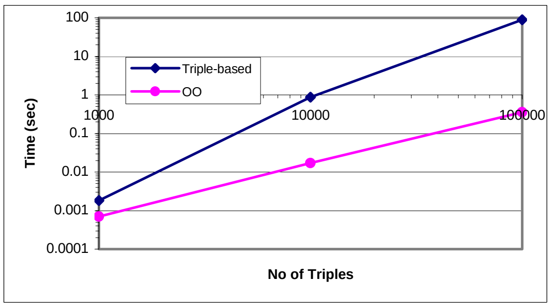


Figure 22. Performance results for Rule Case 1.

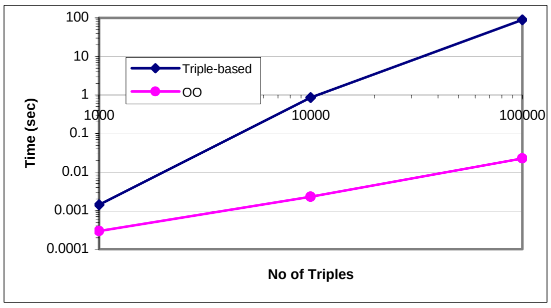


Figure 23. Performance results for Rule Case 2.

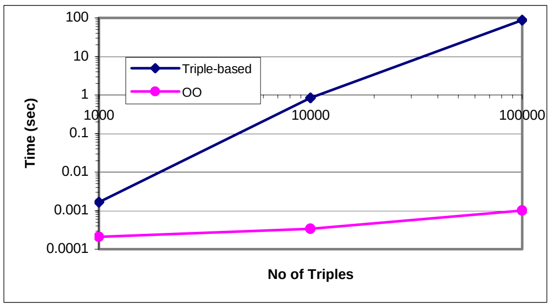


Figure 24. Performance results for Rule Case 3.

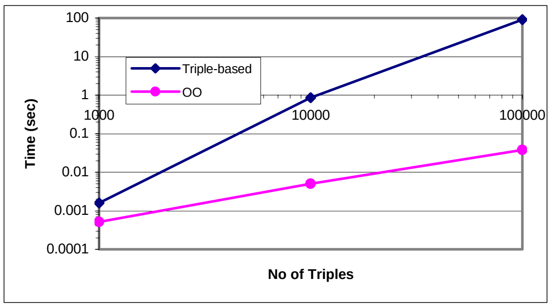


Figure 25. Performance results for Rule Case 4.

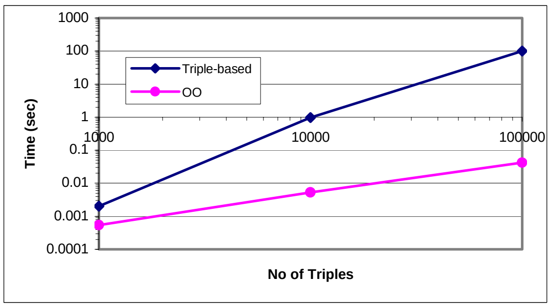


Figure 26. Performance results for Rule Case 5.

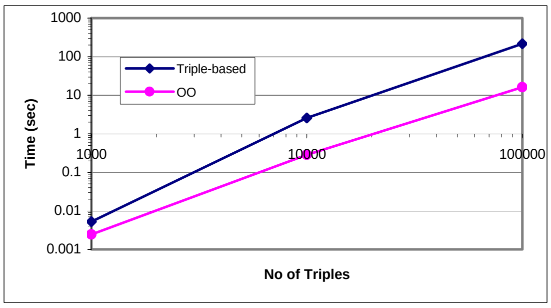


Figure 27. Performance results for Rule Case 6.

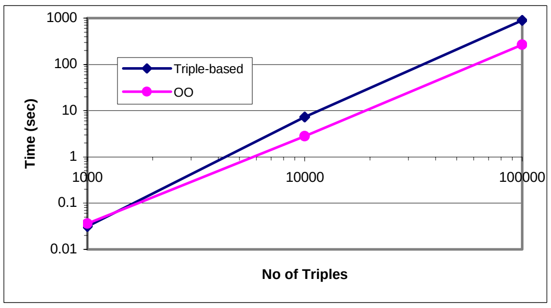


Figure 28. Performance results for Rule Case 7.

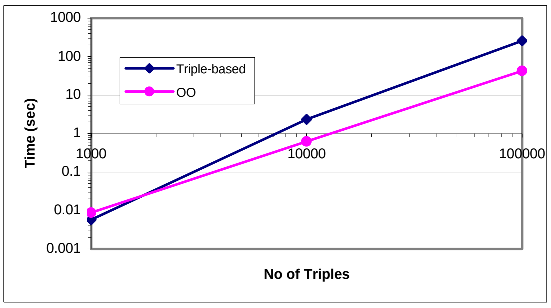


Figure 29. Performance results for Rule Case 8.

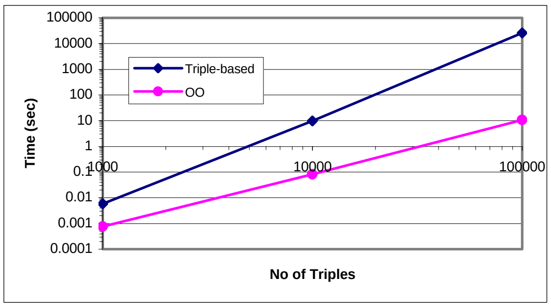


Figure 30. Performance results for Rule Case 9.

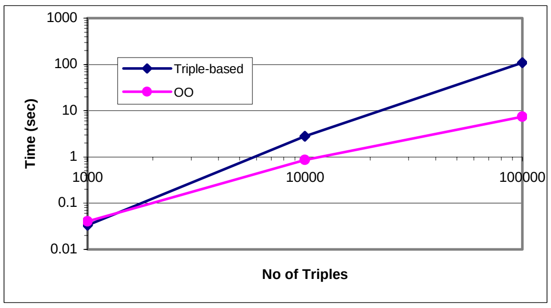
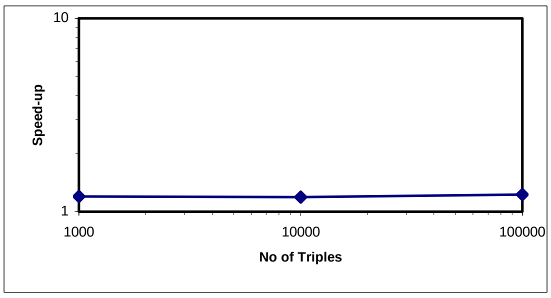


Figure 31. Performance results for Rule Case 10.



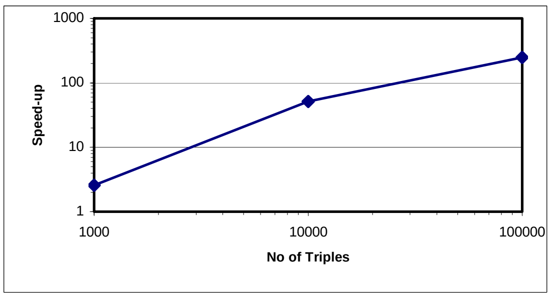


Figure 32. Speed-up of Rule Case 0.

Figure 33. Speed-up of Rule Case 1.

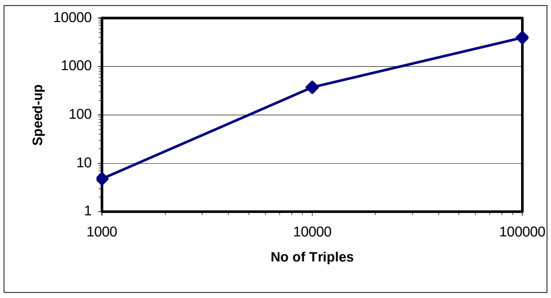


Figure 34. Speed-up of Rule Case 2.

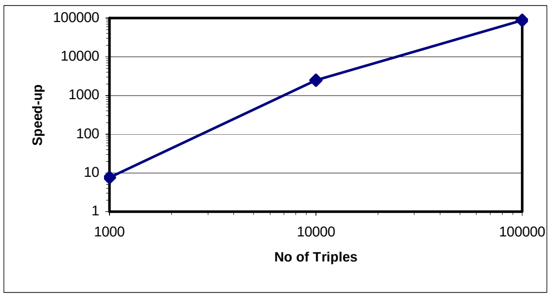


Figure 35. Speed-up of Rule Case 3.

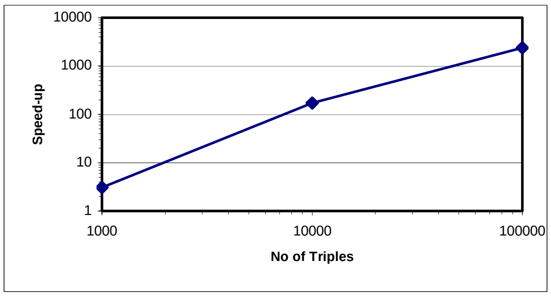


Figure 36. Speed-up of Rule Case 4.

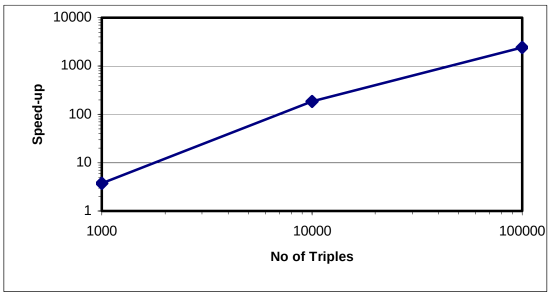


Figure 37. Speed-up of Rule Case 5.

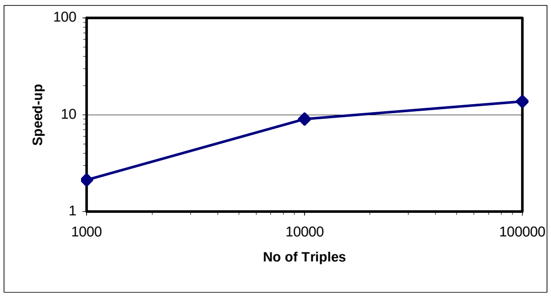


Figure 38. Speed-up of Rule Case 6.

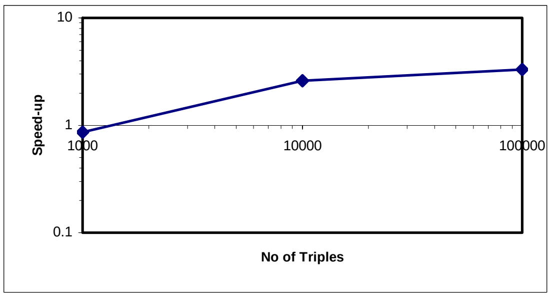


Figure 39. Speed-up of Rule Case 7.

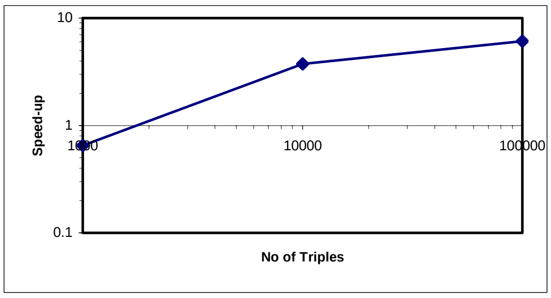


Figure 40. Speed-up of Rule Case 8.

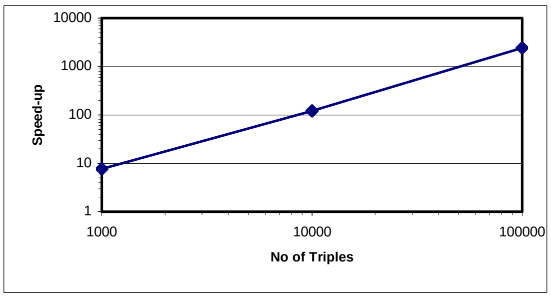


Figure 41. Speed-up of Rule Case 9.

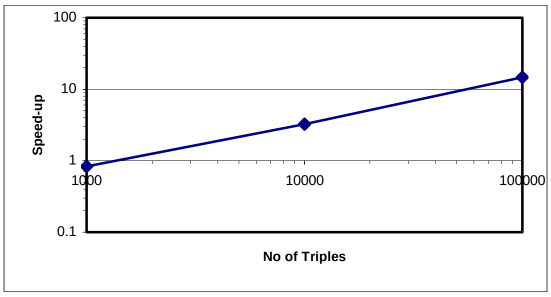


Figure 42. Speed-up of Rule Case 10.

Finally, Table 10 shows the performances of R-DEVICE when all rule cases are simultaneously fed into the system, in order to test whether the performance is compromised by the presence of multiple rules. Table 10 also includes the theoretical performance time for all rules which is calculated by the sum of the times of each individual rule case. Results clearly show that running many rules simultaneously is more or less the same as running many rules individually, i.e. the performance results of R-DEVICE is not compromised by the presence of multiple rules.

Table 10. Performance Results for all Rule Cases.

Number of triples	Theoretical	Measured
1K	0.0974	0.0945
10K	4.8085	4.6154
100K	358.1527	367.9670

The major improvement in performance of the object-oriented approach of R-DEVICE compared to the triple-based approach can be attributed to two reasons:

- An OO-RDF rule retrieves values of different slots of the same object in one step, while triple queries need to perform a join even for properties of the same subject. This large number of joins, which are proportional to the number of triples used in the rule, is the main reason for the worse performance of triple-based rules in almost all of the cases. In rule case 0 the two approaches perform almost the same regardless of the number of triples because the rule condition involves only one property of only one resource, i.e. a single triple.
- An OO-RDF rule ranges over fewer objects because objects are "clustered" by class. On the other hand, triple queries always range over the whole set of triples which are instances of the triple class.

In very few cases, the triple based approach performs better than the OO approach. This happens only when the number of triples is small (1000) and is attributed to the fact that there are certain start-up costs that pay-off when the number of objects is large. However, even in these cases the difference in performance is very small.

The performance of both approaches benefits from the use of the RETE algorithm for matching the rule conditions. The performance of the triple-based RDF model is usually boosted by the use of indices (e.g. [1], [41]), but so could the OO-RDF model. We believe that the first reason above is a generic difference between OO- and triple-based RDF models that leads to such a big performance difference.

Finally, we include a comparison of R-DEVICE with the XPCE Semantic Web library of SWI-Prolog [41], which is a triple-based RDF storage/inferencing system that uses indices to boost its performance. All rule cases have been implemented in SWI-Prolog and their performance has been measured on the same machine with R-DEVICE. Results are included in Table 11. Figures 43-53 display the comparative performance scale-up for all rule cases.

Results show that for some rule cases, i.e. 0, 1, 2, 3, 9, and 10, R-DEVICE is faster, while for the rest of the rule cases, i.e. 4, 5, 6, 7, and 8, XPCE is faster. More importantly, for rule cases 1, 2, 3, and 10, R-DEVICE scales-up better than XPCE, while R-DEVICE does not scale-up worse than XPCE in any case. The explanation for the inferior performance of R-DEVICE in some rule cases is that the use of indices in these cases reduces the complexity of the rule condition more than the use of objects and RETE. However, as we have mentioned above, the OO approach could also benefit from using indices for matching object slots. Furthermore, the use of indices is not free; in [41] it is reported that indices require memory space that is almost twice as large as the memory space occupied by the RDF triples. The explanation for the worse scale-up of XPCE in some rule cases is the use of the backtracking mechanism that generates many unnecessary variable-value combinations. Furthermore, in rule case 10 that recursion is used to generate intermediate auxiliary results, R-DEVICE materializes these results, so they are created once and used many times, whereas XPCE re-calculates them each time they are needed.

Table 11. Performance Results for SWI-Prolog/XPCE Semantic Web Library

Rule Case	**No of triples**		
	1K	10K	1000K
0	0.03437	3.73400	280.00000
1	0.00041	0.07703	9.68800
2	0.00015	0.00297	0.43910
3	0.00011	0.00091	0.01109
4	0.00016	0.00102	0.01375
5	0.00025	0.00106	0.01390
6	0.00088	0.15140	3.98500
7	0.03203	1.25000	191.76500
8	0.00098	0.03890	4.70300
9	0.00071	0.13130	23.09400
10	0.05750	1.92200	219.65700

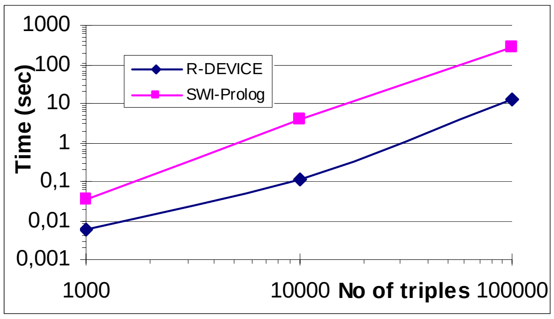


Figure 43. Comparison with SWI-Prolog for Rule Case 0.

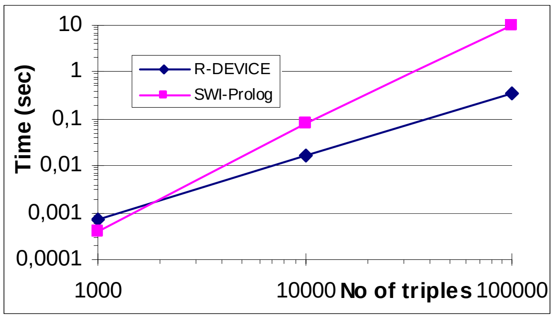


Figure 44. Comparison with SWI-Prolog for Rule Case 1.

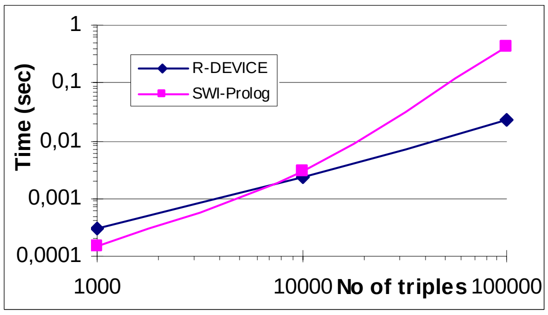


Figure 45. Comparison with SWI-Prolog for Rule Case 2.

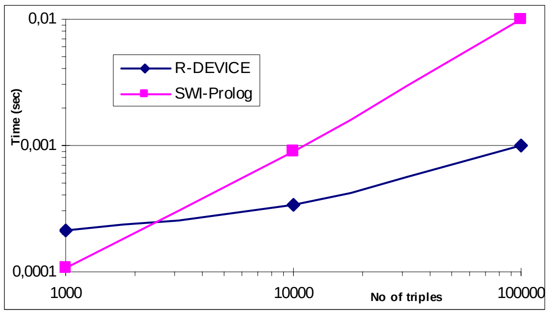


Figure 46. Comparison with SWI-Prolog for Rule Case 3.

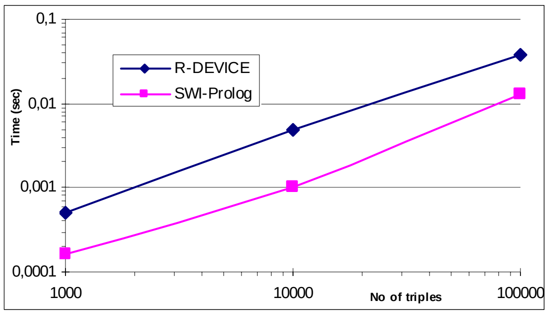


Figure 47. Comparison with SWI-Prolog for Rule Case 4.

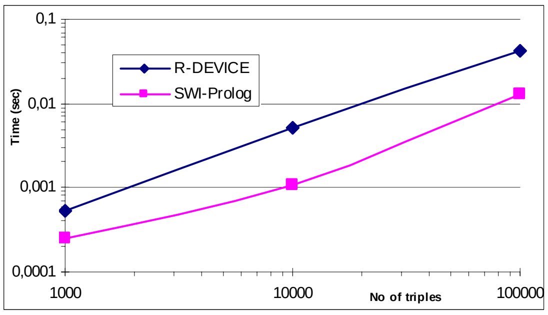


Figure 48. Comparison with SWI-Prolog for Rule Case 5.

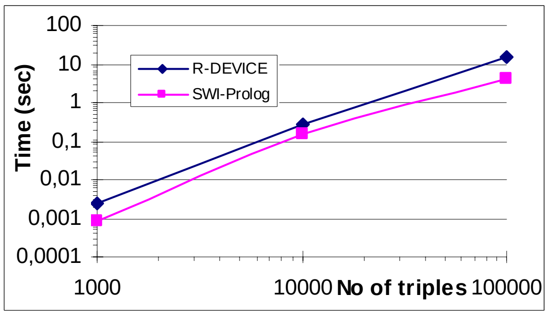


Figure 49. Comparison with SWI-Prolog for Rule Case 6.

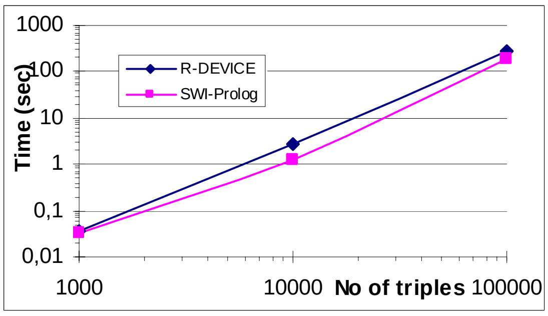


Figure 50. Comparison with SWI-Prolog for Rule Case 7.

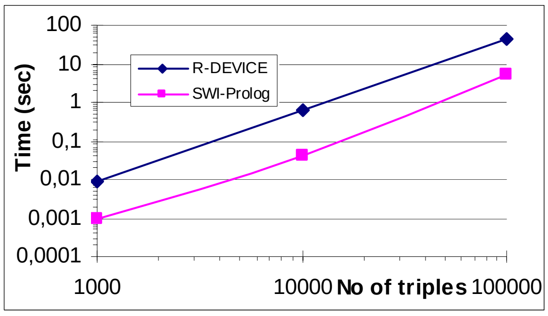


Figure 51. Comparison with SWI-Prolog for Rule Case 8.

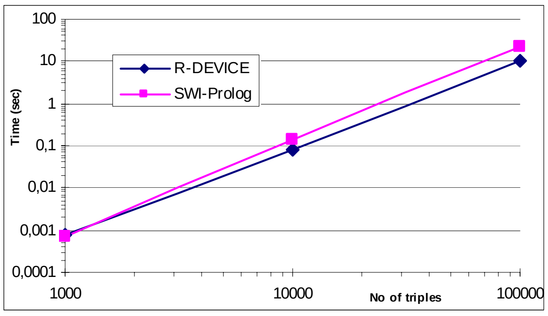


Figure 52. Comparison with SWI-Prolog for Rule Case 9.

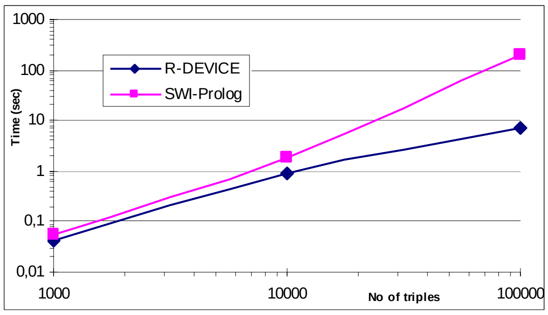


Figure 53. Comparison with SWI-Prolog for Rule Case 10.

6 Related Work

Many RDF rule languages exist in the literature. Some on-line surveys of RDF Inference and Query systems can be found in [37] and [36]. In this section, we will refer to some of the most representative ones and we will compare them to R-DEVICE.

TRIPLE [38], an extension of the SiLRI system [18], is an RDF rule (query, inference, and transformation) language, with a layered and modular nature, that is based on Horn Logic and F-Logic and aims to support applications in need of RDF reasoning and transformation, i.e., to provide mechanisms to query web resources in a declarative way. However, in contrast with many other RDF rule/query languages, TRIPLE allows the semantics of languages on top of RDF to be defined with rules, instead of supporting the same functionality with built-in semantics. Wherever the definition of language semantics is not easily possible with rules (e.g., OWL [33]), TRIPLE provides access to external programs, like description logic classifiers.

TRIPLE permits the usage of path expressions, but not generalized path expressions, i.e. the path length and composition must be entirely known. Furthermore, compared to R-DEVICE, TRIPLE does not support aggregate, grouping, sorting and user-defined functions. Rules in TRIPLE are used for transient querying and cannot be used for defining and maintaining views. As its name implies, the query and data model of TRIPLE is triples, therefore TRIPLE requires multiple joins for collecting all the properties of a resource, since property instances and resource instances are stored in different database relations (or in different tuples of the same relation). In this paper we have shown that the OO-RDF data model of R-DEVICE is superior in performance compared to the triple-based data model of most RDF query and rule languages. Finally, TRIPLE does not have a RuleML compatible syntax.

SweetJess [24] is an implementation of a defeasible reasoning system (situated courteous logic programs) based on Jess. R-DEVICE is a deductive rule language that supports non-monotonicity in terms of negation-as-failure. Furthermore, recently we have developed a defeasible logic extension to R-DEVICE [3]. SweetJess integrates well with RuleML, as does R-DEVICE. However, SweetJess rules can only express reasoning over ontologies expressed in DAMLRuleML (a DAML+OIL like syntax of RuleML) and not on arbitrary RDF data, like R-DEVICE. Furthermore, SweetJess is restricted to simple terms (variables and atoms). R-DEVICE can support a limited form of functions in the following sense: (a) path expressions are allowed in the rule condition, which can be seen as complex functions, where allowed function names are object referencing slots; (b) aggregate and sorting functions are allowed in the conclusion of aggregate rules. Finally, R-DEVICE can also support conclusions in non-stratified rule programs due to the presence of truth-maintenance rules.

SWRL [27] is a rule language based on a combination of the OWL DL and Lite sublanguages of OWL [33] with the Unary/Binary Datalog sublanguages of RuleML [14]. SWRL enables Horn-like rules to be combined with an OWL knowledge base. SWRL provides several types of syntaxes, including RuleML and RDF-like. SWRL is also based on the triple model of RDF and is a first-order logic language specification. Negation is not explicitly supported by the SWRL language, but only indirectly through OWL DL (e.g. class complements). There is a concrete implementation of SWRL, called Hoolet[5]. Hoolet is an implementation of an OWL-DL reasoner that uses a first order prover. The ontology is translated 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.

The Edutella project [35] provides a family of Datalog like languages, called RDF-QEL-i, that support different levels of query capabilities among distributed, heterogeneous RDF repositories. The highest level language RDF-QEL-5 is equivalent to stratified Datalog. Furthermore, aggregation and foreign functions are supported.

[5] `http://owl.man.ac.uk/hoolet/`

Actually, the RDF-QEL-i languages provide a common query and inference syntax and semantics for the heterogeneous peers and are translated into the base rule/query language of each peer. Several query language wrappers have been implemented, such as RQL, TRIPLE, etc. The common data model of Edutella is based on triples and an RDF-like syntax is provided. Path expressions and view maintenance are not supported.

CWM [11] is a general-purpose data processor for the semantic web. It is a forward chaining reasoner which can be used for querying, checking, transforming and filtering information. Its core language is RDF, extended to include rules, and it uses RDF/XML or RDF/N3. CWM supports path expressions, like TRIPLE, but only concrete ones, i.e. the path length should be known in advance and every step in the path should be ground. Furthermore, CWM does not support negation. CWM allows aggregated functions but not grouping and sorting.

Jena [32] is based on the RDF triple data model and has an inference subsystem that allows a range of inference engines or reasoners to be plugged into Jena. The inference mechanism is designed to be quite general and includes a generic rule engine that can be used for many RDF processing or transformation tasks. The generic rule reasoner supports user defined rules under forward chaining, tabled backward chaining and hybrid execution strategies. The rule language allows a limited form of functors, but does not support negation, aggregation or path expressions. Jena rules do not have a RuleML-like syntax, but the extensibility of the system allows for different syntaxes. Finally, the Jena rule system allows maintenance of asserted conclusions, which however is trivial due to the lack of negation.

Sesame [16] supports inferencing through a forward chaining inferencer that does a pruned iterative sweep over the store, computes the closure and stores it in the repository. Thus, at query time, every inferencing task is reduced to a simple database lookup. Apart from the default RDF/RDFS semantics, the user is allowed to specify his/her own entailment rules in an XML-based rule file. However, although their forward chaining algorithm is optimized for a fixed set of entailment rules, it is not clear whether it is still efficient for an arbitrary number of user-defined rules. Furthermore, since RDF/RDFS entailment rules do not have negation, it is unclear how Sesame handles negation in the rule condition.

ROWL [20] is a system that enables users to express rules in RDF/XML syntax using an ontology in OWL. Using XSLT stylesheets, the rules in RDF/XML are transformed into forward-chaining production rules in JESS. ROWL also uses stylesheets to transform ontology and instance files into Jess unordered triple facts, which is also the model followed by the rules. ROWL does not maintain the assertions derived by the rules and does not support either negation, path expressions or aggregate functions. The rule language has been used in a Semantic Web environment for pervasive computing where agents reason about context and privacy concerns of the user [21].

Bossam [28] is a RETE-based forward-chaining production rule engine that has an RDF logic-like rule language, called Buchingae. Bossam has an RDF-like knowledge representation scheme and supports both strong and weak negation and second-order typed predicates. The Bossam data model is based on triples, therefore second order syntax is actually translated into first-order querying over property and/or type definition triples. Negation is supported by the rule language; however, no hint on how it is implemented by the rule engine is given. Bossam also provides a RuleML-like language, called LogicML, which however overrides several of the

RuleML elements, hindering compatibility with standard RuleML. Finally, inference results exported by Bossam are flat, i.e. there is no notion of derived classes and properties.

7 Conclusions and Future Work

In this paper, we have presented a deductive object-oriented knowledge base system, called R-DEVICE, which imports RDF data into the CLIPS production rule system as COOL objects and uses a deductive rule language for querying and reasoning about them.

The main difference between the RDF triple-based model and our OO-RDF model is that we treat properties mainly as attributes encapsulated inside resource objects, as in traditional OO programming languages. In this way properties about a single resource are gathered together in one object, resulting in superior inference/query performance compared to the performance of a triple-based model, as it was experimentally shown in this paper. Another reason for better performance of our OO model is that objects are clustered by class; therefore, rule conditions range over fewer objects. On the other hand, in the triple-based model rules always range over the whole set of triples.

R-DEVICE features a powerful deductive rule language which is able to draw inferences both on the RDF schema and data. The rule language includes features such as ground and generalized path expressions, stratified negation, aggregate, grouping, and sorting functions. All these can be combined with a second-order syntax, where variables can range over classes and properties, which is safely and efficiently translated into first-order syntax at compile-time. Users can define materialized views with R-DEVICE rules which are incrementally maintained by truth-maintaining CLIPS production rules. Finally, users can choose between a native CLIPS-like syntax and a RuleML-like syntax.

Although R-DEVICE is implemented in an environment where the Closed-World Assumption has a strong "tradition", careful design of the RDF transformation and rule compilation algorithms have managed to successfully handle the Open-World Assumption of RDF and the Semantic Web. Concerning the RDF data model, the mismatch between the descriptive semantics of RDF data and the prescriptive semantics of CLIPS call for dynamic redefinitions of resource classes and objects, which are handled by R-DEVICE effectively. All assertions are considered to be true, which is compatible with the Open-World Assumption. Under the Closed-World Assumption some statements would cause a consistency violation error. Such behaviour could be very easily implemented in CLIPS; however, this was a design choice for R-DEVICE. An alternative would be to leave on the user the choice on which assumption to base the RDF transformation.

Another point where the Open-World vs. the Closed-Word Assumption arises in R-DEVICE is in the second-order translation process. The grounding of class and/or property variables is done at rule compile-time considering only the loaded classes and properties (Closed-Word Assumption). However, as discussed in the paper, it is easy to perform an incremental rule compilation upon the loading of new RDF Schema documents (Open-World Assumption). Finally, the semantics of derived attribute and aggregate attribute rules are non-monotonic since the import of new RDF documents causes results to be re-calculated.

We have also developed a defeasible logic extension of R-DEVICE [3] which has been used as a backend reasoning mechanism by negotiating agents to express and apply various negotiation strategies [39]. We are currently working on the development of a visual editor [4] for the RuleML-like rule language, integrated into a visual rule-base development environment. Furthermore, we are extending the system to handle ontologies in OWL [33], partly by extending the triple translator to capture the extended semantics of OWL and partly by extending the rule language translator [34]. Among our future plans is to continue optimizing the performance of R-DEVICE and further testing it using benchmarking suites, such as the Lehigh University Benchmark [25]. Furthermore, we plan deploy R-DEVICE as a Web Service and to develop an interface to RDF storage systems, such as ICS-FORTH RDFSuite [1] or Sesame [16].

Finally, transforming RDF resources into traditional objects has the advantage that RDF data can interoperate seamlessly with other data models, such as the object-oriented or the relational data models. In this way, CLIPS objects can easily be exported as RDF data or RDF data can be easily used in an expert system developed in CLIPS. Furthermore, we believe that the object data model is the most general and expressive one; therefore, it can serve as a mediator between several data models, both traditional and web models. For example, we have also used CLIPS to transform XML documents [9] and OWL ontologies [34] into objects. Among our future plans is to integrate all these translators into a single system and have our deductive rule language to reason over all such web data models in a homogeneous manner. Results could be easily exported in any of these models. Therefore, our system could be used as a translator between these languages.

Acknowledgments

We would like to thank the anonymous reviewers of this paper for their valuable comments. This work was partially supported by the Greek R&D General Secretariat through a bilateral Greek-Ukrainian project (EPAN-M.4.3.6.1.γ, No. 148-γ) and through a PENED program (EPAN M.8.3.1, No. 03EΔ73).

References

[1] Alexaki S., Christophides V., Karvounarakis G., Plexousakis D., and Tolle K., “The ICSFORTH RDFSuite: Managing Voluminous RDF Description Bases”, *Proc. 2nd Int. Workshop on the Semantic Web*, pp. 1-13, Hong Kong, 2001.

[2] Antoniou G., Wagner G., "Rules and Defeasible Reasoning on the Semantic Web", in *Proc. RuleML Workshop 2003*, Springer-Verlag, LNCS 2876, pp. 111–120, 2003.

[3] Bassiliades N., Antoniou G., Vlahavas I., “A Defeasible Logic Reasoner for the Semantic Web”, *International Journal on Semantic Web and Information Systems*, Vol. 2, No. 1, pp. 1-41, 2006.

[4] Bassiliades N., Kontopoulos E., Antoniou G., “A Visual Environment for Developing Defeasible Rule Bases for the Semantic Web”, *Proc. International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML-2005)*, A. Adi, S. Stoutenburg, S. Tabet (Ed.), Springer-Verlag, LNCS 3791, pp. 172-186, Galway, Ireland, 10-12 Nov. 2005.

[5] Bassiliades N., Vlahavas I., “Capturing RDF Descriptive Semantics in an Object Oriented Knowledge Base System”, *Poster Proc. 12th Int. WWW Conf. (WWW2003)*, Budapest.

[6] Bassiliades N., Vlahavas I., “Processing Production Rules in DEVICE, an Active Knowledge Base System”, *Data & Knowledge Engineering*, 24(2), pp. 117-155, 1997.

[7] 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)*, G. Antoniou, H. Boley (Eds.), Springer-Verlag, LNCS 3323, pp. 65-80, Hiroshima, Japan, 2004.

[8] Bassiliades N., Vlahavas I., and Elmagarmid A.K., "E-DEVICE: An extensible active knowledge base system with multiple rule type support", *IEEE TKDE*, 12(5), pp. 824-844, 2000.

[9] Bassiliades N., Vlahavas I., and Sampson D., "Using Logic for Querying XML Data", in *Web-Powered Databases*, Ch. 1, pp. 1-35, Idea-Group Publishing, 2003.

[10] Bassiliades N., Vlahavas I., Elmagarmid A., Houstis E., “Interbase-KB: A Knowledge-based Multidatabase Sustem for Data Warehousing”, *IEEE Transactions on Knowledge and Data Engineering*, 15(5), pp. 1188-1205, 2003.

[11] Berners-Lee T., "CWM - closed world machine", `http://www.w3.org/2000/10/swap/doc/cwm.html`, 2000.

[12] Berners-Lee T., "Web Design Issues: Architectural and philosophical points", `http://www.w3.org/DesignIssues/`

[13] Berners-Lee T., Hendler J., Lassila O., "The Semantic Web", *Scientific American*, May 2001.

[14] 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.

[15] Brickley D. and Guha R.V., "RDF Vocabulary Description Language 1.0: RDF Schema", *W3C Recommendation*, 10 Feb. 2004, `http://www.w3.org/TR/rdf-schema/`

[16] Broekstra J., Kampman A., van Harmelen F., “Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema”, *Proc. 1st Int. Semantic Web Conf.*, Springer-Verlag, LNCS 2342, pp. 54-68, 2002.

[17] *CLIPS Basic Programming Guide*, Version 6.21, June 15th 2003, `http://www.ghg.net/clips/Download.html`

[18] Decker S., Brickley D., Saarela J., Angele J., "A query and inference service for RDF", in *QL’98 - The Query Languages Workshop*, Boston, USA, 1998.

[19] Diaz O., Jaime A., "EXACT: An Extensible Approach to Active Object-Oriented Databases", *VLDB Journal*, 6(4), pp. 282-295, 1997.

[20] 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`

[21] Gandon F., Sadeh N., “Semantic Web Technologies to Reconcile Privacy and Context Awareness”, *Web Semantics Journal*, Vol. 1, No. 3, 2004.

[22] Grant J., Beckett D., "RDF Test Cases", W3C Recommendation, Feb. 2004. `http://www.w3.org/TR/rdf-testcases/`

[23] Gray P.M.D., Kulkarni K.G., and Paton N.W., *Object-Oriented Databases, A Semantic Data Model Approach*, Prentice Hall, London, 1992.

[24] Grosof B.N., Gandhe M.D., Finin T.W., “SweetJess: Translating DAMLRuleML to JESS”, *Proc. RuleML Workshop*, 2002.

[25] Guo Y., Pan Z., Heflin J., "LUBM: A benchmark for OWL knowledge base systems", *Web Semantics: Science, Services and Agents on the World Wide Web*, Vol. 3, 2005, pp. 158–182.

[26] Hayes P., "RDF Semantics", *W3C Recommendation*, Feb. 2004. `http://www.w3.org/TR/rdf-mt/`

[27] Horrocks I., Patel-Schneider P. F., Boley H., Tabet S., Grosof B., Dean M., "SWRL: A Semantic Web Rule Language Combining OWL and RuleML", W3C Member Submission, May 2004. `http://www.w3.org/Submission/2004/03/`

[28] Jang M., "Bossam - A Java-based Rule Processor for the Semantic Web", `http://mknows.etri.re.kr/bossam`

[29] Kifer M., Lausen G., "F-Logic: A Higher-Order language for Reasoning about Objects, Inheritance, and Scheme", *SIGMOD Conference*, 1989, pp. 134-146.

[30] Magkanaraki A., Karvounarakis G., Anh T.T., Christophides V., Plexousakis D., “Ontology Storage and Querying”, *TR 308, ICS-FORTH*, Greece, April 2002, `http://139.91.183.30:9090/RDF/publications/tr308.pdf`

[31] Manola F., Miller E., "RDF Primer", W3C Recommendation, Feb 2004. `http://www.w3.org/TR/rdf-primer/`

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

[33] McGuinness D. L., van Harmelen F., "OWL Web Ontology Language: Overview", W3C Recommendation, Feb 2004. `http://www.w3.org/TR/owl-features/`

[34] Meditskos G., Bassiliades N., “Towards an Object-Oriented Reasoning System for OWL”, *Int. Workshop on OWL Experiences and Directions*, Nov. 2005, Galway, Ireland, 2005. `http://www.mindswap.org/2005/OWLWorkshop/programme.shtml`

[35] Nejdl W., Wolf B., Qu C., Decker S., Sintek M., Naeve A., Nilsson M., Palmer M., Risch T., "Edutella: A P2P networking infrastructure based on RDF", in *Proc. of WWW-2002*, ACM Press, 2002, pp. 604-615.

[36] Prud'hommeaux E., “RDF Query and Rules Status”, `http://www.w3.org./2001/11/13-RDF-Query-Rules/`

[37] Seaborne A., and Reggiori A., "RDF Query and Rule languages Use Cases and Examples survey", `http://rdfstore.sourceforge.net/2002/06/24/rdf-query/`

[38] Sintek M., Decker S., “TRIPLE-A Query, Inference, and Transformation Language for the Semantic Web”, *Proc. 1st Int. Semantic Web Conf.*, Springer-Verlag, LNCS 2342, pp. 364-378, 2002.

[39] Skylogiannis T., Antoniou G., Bassiliades N., Governatori G., “DR-NEGOTIATE – A System for Automated Agent Negotiation with Defeasible Logic-Based Strategies”, *IEEE International Conference on E-Technology, E-Commerce and E-Service*, W.K.W. Cheung, J. Hsu (Ed.), IEEE, pp. 44-49, 29/3 - 1/4/2005, Hong Kong, China, 2005.

[40] Ullman J., *Principles of Database and Knowledge-Base Systems*, Rockville, Maryland: Computer Science Press, 1989.

[41] Wielemaker J., Schreiber G., Wielinga B., "Prolog-Based Infrastructure for RDF: Scalability and Performance", *Proc. 2nd Int. Semantic Web Conf.*, pp. 644 - 658, LNCS 2870, Springer-Verlag, 2003.

[42] Xalan-Java XSLT processor, `xml.apache.org/xalan-j/`

Appendix A. R-DEVICE Syntax

This appendix contains the syntax of R-DEVICE rules in BNF notation as an extension of CLIPS rules.

```
<r-device-rule> ::= <deductive-rule> | <derived-attribute-rule> | <aggregate-attribute-rule>

<deductive-rule> ::=
   (deductiverule [<rule-name>]
     <conditional-element>*
   =>
     <conclusion>)

<derived-attribute-rule> ::=
   (derivedattrule [<rule-name>]
     <conditional-element>*
   =>
     <derived-attribute-conclusion>)

<aggregate-attribute-rule> ::=
   (aggregateattrule [<rule-name>]
     <conditional-element>*
   =>
     <aggregate-attribute-conclusion>)

<conditional-element> ::= <pattern-CE> | <assigned-pattern-CE> | <not-CE> |
                              <and-CE> | <or-CE> | <test-CE>

<pattern-CE> ::= <class-pattern-CE>

<assigned-pattern-CE> ::= <single-field-variable> <- <pattern-CE> |
                              <instance-name> <- <pattern-CE>

<not-CE> ::= (not <conditional-element>)

<and-CE> ::= (and <conditional-element>+)

<or-CE> ::= (or <conditional-element>+)

<test-CE> ::= (test <function-call>)

<class-pattern-CE> ::= (<class-expr> <LHS-slot>*)

<class-expr> ::= <class-name> | <svar-expr> | <namespace>':'<class-name> |
                   <svar-expr>':'<class-name> | <namespace>':'<svar-expr>

<LHS-slot> ::= <single-field-LHS-slot> | <multifield-LHS-slot>

<single-field-LHS-slot> ::= (<path-expr> <constraint>)

<multifield-LHS-slot> ::= (<path-expr> <constraint>*)

<path-expr> ::= <slot-expr> | (<path-item>+)

<slot-expr> ::= <slot-name> | <svar-expr>

<svar-expr> ::= <single-field-variable> | '?'

<path-item> ::= <slot-expr> | <multifield-variable> | (<slot-name>+)

<constraint> ::= '?' | '$?' | <connected-constraint>

<connected-constraint> ::= <single-constraint> |
                              <single-constraint> '&' <connected-constraint> |
                              <single-constraint> '|' <connected-constraint>

<single-constraint> ::= <term> | ~<term>

<term> ::= <constant> | <single-field-variable> | <multifield-variable> | ':'<function-call> |
            '='<function-call> | <single-field-variable-multifield-expression>

<single-field-variable> ::= '?'<variable-symbol>

<multifield-variable> ::= '$?'<variable-symbol>

<single-field-variable-multifield-expression> ::= '??'<variable-symbol>

<constant> ::= <symbol> | <string> | <integer> | <float> | <instance-name>
```

```
<function-call> ::= (<function-name> <expression>*)

<conclusion> ::= [(calc <function-call>+)] (<RHS-class-expr> <RHS-slot>*)

<RHS-class-expr> ::= <class-name> | <single-field-variable> | <namespace>':'<class-name> |
              <single-field-variable>':'<class-name> | <namespace>':'<single-field-variable>

<RHS-slot> ::= <simple-assign-expr> | <aggregate-assign-expr>

<simple-assign-expr> ::= (<RHS-slot-expr> <value>)

<RHS-slot-expr> ::= <slot-name> | <single-field-variable>

<value> ::= <single-field-variable> | <multifield-variable> | <constant>

<aggregate-assign-expr> ::= (<RHS-slot-expr> <aggregate-function-expr>)

<aggregate-function-expr> ::= (<aggregate-function> <single-field-variable>)

<derived-attribute-conclusion> ::= [(calc <function-call>+)]
                   <single-field-variable> <- (<RHS-class-expr> <simple-assign-expr>)

<aggregate-attribute-conclusion> ::= [(calc <function-call>+)]
                 <single-field-variable> <- (<RHS-class-expr> <aggregate-assign-expr>)
```

`<rule-name> ::=` *A symbol which represents the name of a rule*

`<variable-symbol> ::=` *A symbol beginning with an alphabetic character.*

`<function-name> ::=` *Any symbol which corresponds to a system or user defined function, a deffunction name, or a defgeneric name*

`<class-name> ::=` *A valid defclass name*

`<slot-name> ::=` *A valid defclass slot name*

`<aggregate-function> ::=` *A valid aggregate function name*

Appendix B. Examples of R-DEVICE rules

This appendix contains examples of R-DEVICE rules for sample RDF queries that have been obtained from [37].

```
(deductiverule q1a
  ?x <- (? (email:message-id '123456@example.com'))
  =>
  (result (email ?x)))


(deductiverule q1b
  ?msg <- (pop3:Message (pop3:property ?prop))
  ?prop <- (? (rdfs:label 'From') (rdf:value ?from & :(str-index "hotmail" ?from)))
  =>
  (result (email ?msg)))


(deductiverule q2
  ?x <- (? (vcard:N ?y))
  ?y <- (? (vcard:Family "Smith") (vcard:Given ?v))
  =>
  (person (name ?v)))


(deductiverule q3
  data:x <- (? (?property ?value))
  ?property <- (rdf:Property (rdfs:range $? ?t $?))
  =>
  (result (property ?property) (value ?value) (type ?t)))


(deductiverule q4
  ?Header <- (hdr:HeaderField (hdr:fieldName ?name) (rdfs:label ?purpose) (hdr:protocol ?p))
  ?p <- (? (hdr:protocolName ?pn) (hdr:spec ?ps))
  ?ps <- (? (hdr:document ?psdocument))
  =>
  (result (header ?Header) (name ?name) (protocol ?p) (purpose ?purpose)
          (pname ?pn) (spec ?ps) (document ?psdocument))
)


(deductiverule q5
  (rss:item (rss:title ?title) (rss:link ?link))
  =>
  (result (title ?title) (link ?link)))


(deductiverule q6
  (rss:item (rss:title ?title & :(str-index "RDQL" ?title)) (rss:link ?link))
  =>
  (result (link ?link)))


(deductiverule q7
  ?x <- (? (dc:title ?tt) (dc:description ?dd) ((etbthes:ETBT dc:subject) ?ss2)
           (dc:identifier ?identifier) ((dcq:RFC1766 dc:language) ?language))
  ?tt <- (? (rdf:value ?t_val) ((dcq:RFC1766 dc:language) ?t_lang))
  ?ss2 <- (? (rdf:value ?subject_val) ((dcq:RFC1766 dc:language) ?subj_lang))
  ?dd <- (? (rdf:value ?desc_val) ((dcq:RFC1766 dc:language) ?desc_lang))
  =>
  (result (title_value ?t_val) (title_language ?t_lang) (subj_val ?subject_val)
          (subj_lang ?subj_lang) (desc_value ?desc_val) (desc_lang ?desc_lang)
          (language ?language)(identifier ?identifier)))
```

Appendix C. R-DEVICE rules for Querying ODP

This appendix contains the R-DEVICE rules that are used in Section 5 for querying ODP metadata and measuring the performance of the object-oriented RDF data model of R-DEVICE against the triple-based RDF data model.

Rule Case 0: Retrieve the title of all resources.

```
(deductiverule oo-rule-0
     (? (dc:title ?t))
  =>
     (result (title ?t)))

(deductiverule triple-rule-0
     (rdf-triple (subject ?x) (predicate [dc:title]) (object ?t))
  =>
     (result (title ?t)))
```

Rule Case 1: Retrieve the title of all `dmoz:Topic` resources.

```
(deductiverule oo-rule-1
     (dmoz:Topic (dc:title ?t))
  =>
     (result (title ?t)))

(deductiverule triple-rule-1
     (rdf-triple (subject ?x) (predicate [rdf:type]) (object [dmoz:Topic]))
     (rdf-triple (subject ?x) (predicate [dc:title]) (object ?t))
  =>
     (result (title ?t)))
```

Rule Case 2: Retrieve the title and associated newsgroups of all topics that have at least one associated newsgroup.

```
(deductiverule oo-rule-2
     (dmoz:Topic (dc:title ?t) (dmoz:newsGroup $? ?n $?))
  =>
     (result (title ?t) (news ?n)))

(deductiverule triple-rule-2
     (rdf-triple (subject ?x) (predicate [rdf:type]) (object [dmoz:Topic]))
     (rdf-triple (subject ?x) (predicate [dc:title]) (object ?t))
     (rdf-triple (subject ?x) (predicate [dmoz:newsGroup]) (object ?n))
  =>
     (result (title ?t) (news ?n)))
```

Rule Case 3: Retrieve the title of a topic with a specific catalog ID.

```
(deductiverule oo-rule-3
     (dmoz:Topic (dmoz:catid "24") (dc:title ?t))
  =>
     (result (title ?t)))

(deductiverule triple-rule-3
     (rdf-triple (subject ?x) (predicate [rdf:type]) (object [dmoz:Topic]))
     (rdf-triple (subject ?x) (predicate [dc:title]) (object ?t))
     (rdf-triple (subject ?x) (predicate [dmoz:catid]) (object "24"))
  =>
     (result (title ?t)))
```

Rule Case 4: Retrieve the title of a topic with a specific catalog ID, along with the titles of all associated pages. The topic must have at least one associated page.

```
(deductiverule oo-rule-4
     (dmoz:Topic (dmoz:catid "24") (dc:title ?t) (dmoz:link $? ?l $?))
     ?l <- (dmoz:ExternalPage (dc:title ?lt))
  =>
     (result (title ?t) (link_title ?lt)))

(deductiverule triple-rule-4
     (rdf-triple (subject ?x) (predicate [rdf:type]) (object [dmoz:Topic]))
     (rdf-triple (subject ?x) (predicate [dc:title]) (object ?t))
     (rdf-triple (subject ?x) (predicate [dmoz:catid]) (object "24"))
     (rdf-triple (subject ?x) (predicate [dmoz:link]) (object ?l))
     (rdf-triple (subject ?l) (predicate [dc:title]) (object ?lt))
  =>
     (result (title ?t) (link_title ?lt)))
```

Rule Case 5: Retrieve the title of a topic with a specific catalog ID, along with the titles and descriptions of all associated pages. The topic must have at least one associated page.

```
(deductiverule oo-rule-5
     (dmoz:Topic (dmoz:catid "24") (dc:title ?t) (dmoz:link $? ?l $?))
     ?l <- (dmoz:ExternalPage (dc:title ?lt) (dc:description ?d))
  =>
     (result (title ?t) (link_title ?lt) (link_desc ?d)))

(deductiverule triple-rule-5
     (rdf-triple (subject ?x) (predicate [rdf:type]) (object [dmoz:Topic]))
     (rdf-triple (subject ?x) (predicate [dc:title]) (object ?t))
     (rdf-triple (subject ?x) (predicate [dmoz:catid]) (object "24"))
     (rdf-triple (subject ?x) (predicate [dmoz:link]) (object ?l))
     (rdf-triple (subject ?l) (predicate [rdf:type]) (object [dmoz:ExternalPage]))
     (rdf-triple (subject ?l) (predicate [dc:title]) (object ?lt))
     (rdf-triple (subject ?l) (predicate [dc:description]) (object ?d))
  =>
     (result (title ?t) (link_title ?lt) (link_desc ?d)))
```

Rule Case 6: Retrieve the titles of all topics other than a topic with a specific catalog ID, along with the titles and descriptions of all associated pages of the ".net" domain. The selected topics must have at least one associated page.

```
(deductiverule oo-rule-6
     (dmoz:Topic (dmoz:catid ~"1") (dc:title ?t) (dmoz:link $? ?l $?))
     ?l <- (dmoz:ExternalPage (dc:title ?lt) (dc:description ?d)
                                  (uri ?uri&:(str-index ".net" ?uri)))
  =>
     (result (title ?t) (link_title ?lt) (link_desc ?d)))

(deductiverule triple-rule-6
   (rdf-triple (subject ?x) (predicate [rdf:type]) (object [dmoz:Topic]))
   (rdf-triple (subject ?x) (predicate [dc:title]) (object ?t))
   (rdf-triple (subject ?x) (predicate [dmoz:catid]) (object ~"1"))
   (rdf-triple (subject ?x) (predicate [dmoz:link]) (object ?l))
   (rdf-triple (subject ?l&:(str-index ".net" ?l))
                (predicate [rdf:type]) (object [dmoz:ExternalPage]))
   (rdf-triple (subject ?l) (predicate [dc:title]) (object ?lt))
   (rdf-triple (subject ?l) (predicate [dc:description]) (object ?d))
  =>
     (result (title ?t) (link_title ?lt) (link_desc ?d)))
```

Rule Case 7: Retrieve the titles of all topics that have at least one subtopic, along with the titles of the subtopics and the titles of all their associated pages. The selected subtopics must have at least one associated page.

```
(deductiverule oo-rule-7
     (dmoz:Topic (dc:title ?top) (dmoz:narrow $? ?n $?))
     ?n <- (dmoz:Topic (dc:title ?t) (dmoz:link $? ?l $?))
     ?l <- (dmoz:ExternalPage (dc:title ?lt))
  =>
     (result (top_title ?top) (title ?t) (link_title ?lt)))

(deductiverule triple-rule-7
   (rdf-triple (subject ?x) (predicate [rdf:type]) (object [dmoz:Topic]))
   (rdf-triple (subject ?x) (predicate [dc:title]) (object ?top))
   (rdf-triple (subject ?x) (predicate [dmoz:narrow]) (object ?n))
   (rdf-triple (subject ?n) (predicate [rdf:type]) (object [dmoz:Topic]))
   (rdf-triple (subject ?n) (predicate [dc:title]) (object ?t))
   (rdf-triple (subject ?n) (predicate [dmoz:link]) (object ?l))
   (rdf-triple (subject ?l) (predicate [rdf:type]) (object [dmoz:ExternalPage]))
   (rdf-triple (subject ?l) (predicate [dc:title]) (object ?lt))
  =>
     (result (top_title ?top) (title ?t) (link_title ?lt)))
```

Rule Case 8: Retrieve the titles of all topics that have at least one subtopic, along with the titles of the subtopics and the titles of all their associated pages of the ".net" domain. The selected subtopics must have at least one associated page.

```
(deductiverule oo-rule-8
     (dmoz:Topic (dc:title ?top) (dmoz:narrow $? ?n $?))
     ?n <- (dmoz:Topic (dc:title ?t) (dmoz:link $? ?l $?))
     ?l <- (dmoz:ExternalPage (dc:title ?lt) (uri ?uri&:(str-index ".net" ?uri)))
  =>
     (result (top_title ?top) (title ?t) (link_title ?lt)))

(deductiverule triple-rule-8
   (rdf-triple (subject ?x) (predicate [rdf:type]) (object [dmoz:Topic]))
   (rdf-triple (subject ?x) (predicate [dc:title]) (object ?top))
   (rdf-triple (subject ?x) (predicate [dmoz:narrow]) (object ?n))
   (rdf-triple (subject ?n) (predicate [rdf:type]) (object [dmoz:Topic]))
   (rdf-triple (subject ?n) (predicate [dc:title]) (object ?t))
   (rdf-triple (subject ?n) (predicate [dmoz:link]) (object ?l))
   (rdf-triple (subject ?l&:(str-index ".net" ?l))
                    (predicate [rdf:type]) (object [dmoz:ExternalPage]))
   (rdf-triple (subject ?l) (predicate [dc:title]) (object ?lt))
  =>
     (result (top_title ?top) (title ?t) (link_title ?lt)))
```

Rule Case 9: Find if there is a topic that is a direct subtopic of two different topics. Retrieve the titles of the two super-topics and the title of the sub-topic. Avoid symmetric solutions by checking if the symmetric solution already exists (using negation-as-failure).

```
(deductiverule oo-rule-9
     ?n0 <- (dmoz:Topic (dc:title ?t0) )
     ?n1 <- (dmoz:Topic (dmoz:narrow $? ?n0 $?) (dc:title ?t1))
     ?n2 <- (dmoz:Topic (dmoz:narrow $? ?n0 $?) (dc:title ?t2&~?t1))
     (not (result (topic ?t0) (supertopic1 ?t2) (supertopic2 ?t1)))
  =>
     (result (topic ?t0) (supertopic1 ?t1) (supertopic2 ?t2)))

(deductiverule triple-rule-9
     (rdf-triple (subject ?n0) (predicate [rdf:type]) (object [dmoz:Topic]))
     (rdf-triple (subject ?n0) (predicate [dc:title]) (object ?t0))
     (rdf-triple (subject ?n1) (predicate [rdf:type]) (object [dmoz:Topic]))
     (rdf-triple (subject ?n1) (predicate [dmoz:narrow]) (object ?n0))
     (rdf-triple (subject ?n1) (predicate [dc:title]) (object ?t1))
     (rdf-triple (subject ?n2) (predicate [rdf:type]) (object [dmoz:Topic]))
     (rdf-triple (subject ?n2) (predicate [dmoz:narrow]) (object ?n0))
     (rdf-triple (subject ?n2) (predicate [dc:title]) (object ?t2&~?t1))
     (not (result (topic ?t0) (supertopic1 ?t2) (supertopic2 ?t1)))
  =>
     (result (topic ?t0) (supertopic1 ?t1) (supertopic2 ?t2)))
```

Rule Case 10: Recursively find the titles of all pages indirectly associated with a specific topic through its subtopics, regardless at which depth. This case requires more than one deductive rule. The first two rules recursively retrieve all the subtopics associated with a specific topic, and the third rule retrieves the titles of the associated pages of all the subtopics that were collected by the first two rules.

```
(deductiverule oo-rule-10a
     (dmoz:Topic (dmoz:catid "24") (dc:title ?top) (dmoz:narrow $? ?n $?))
  =>
     (temp-result (top_title ?top) (subtopic ?n)))

(deductiverule oo-rule-10b
     (temp-result (top_title ?top) (subtopic ?n))
     ?n <- (dmoz:Topic (dmoz:narrow $? ?n1 $?))
  =>
     (temp-result (top_title ?top) (subtopic ?n1)))

(deductiverule oo-rule-10c
     (temp-result (top_title ?top) (subtopic ?n))
     ?n <- (dmoz:Topic (dmoz:link $? ?l $?))
     ?l <- (dmoz:ExternalPage (dc:title ?lt) )
  =>
     (result (top_title ?top) (link_title ?lt)))
```

```
(deductiverule triple-rule-10a
     (rdf-triple (subject ?x) (predicate [rdf:type]) (object [dmoz:Topic]))
     (rdf-triple (subject ?x) (predicate [dc:title]) (object ?top))
     (rdf-triple (subject ?x) (predicate [dmoz:catid]) (object "24"))
     (rdf-triple (subject ?x) (predicate [dmoz:narrow]) (object ?n))
  =>
     (temp-result (top_title ?top) (subtopic ?n)))

(deductiverule triple-rule-10b
     (temp-result (top_title ?top) (subtopic ?n))
     (rdf-triple (subject ?n) (predicate [rdf:type]) (object [dmoz:Topic]))
     (rdf-triple (subject ?n) (predicate [dmoz:narrow]) (object ?n1))
  =>
     (temp-result (top_title ?top) (subtopic ?n1)))

(deductiverule triple-rule-10c
     (temp-result (top_title ?top) (subtopic ?n))
     (rdf-triple (subject ?n) (predicate [rdf:type]) (object [dmoz:Topic]))
     (rdf-triple (subject ?n) (predicate [dmoz:link]) (object ?l))
     (rdf-triple (subject ?l) (predicate [rdf:type]) (object [dmoz:ExternalPage]))
     (rdf-triple (subject ?l) (predicate [dc:title]) (object ?lt))
  =>
     (result (top_title ?top) (link_title ?lt)))
```

Appendix D. Sample Interaction with R-DEVICE

This appendix contains a sample interaction with R-DEVICE rules for the RDF document of Figure 8 in section 3.5.

```
CLIPS> (import-rdf "example" local)
Loading namespaces: dmoz example
Inserting: <http://directory.mozilla.org/rdf/Top> <http://directory.mozilla.org/rdf/catid> "1"
.
Inserting: <http://directory.mozilla.org/rdf/Top> <http://purl.org/dc/elements/1.1/title>
"Top" .
Inserting: <http://directory.mozilla.org/rdf/Top> <http://directory.mozilla.org/rdf/narrow>
<http://directory.mozilla.org/rdf/Top/Arts> .
Inserting: <http://directory.mozilla.org/rdf/Top> <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> <http://directory.mozilla.org/rdf/Topic> .
Inserting: <http://directory.mozilla.org/rdf/Top/Arts>
<http://directory.mozilla.org/rdf/catid> "2" .
Inserting: <http://directory.mozilla.org/rdf/Top/Arts> <http://purl.org/dc/elements/1.1/title>
"Arts" .
Inserting: <http://directory.mozilla.org/rdf/Top/Arts> <http://directory.mozilla.org/rdf/link>
<http://www3.bc.sympatico.ca/PHILLIPSHOTGLASS/GlassPage.html> .
Inserting: <http://directory.mozilla.org/rdf/Top/Arts> <http://www.w3.org/1999/02/22-rdf-
syntax-ns#type> <http://directory.mozilla.org/rdf/Topic> .
Inserting: <http://www3.bc.sympatico.ca/PHILLIPSHOTGLASS/GlassPage.html>
<http://purl.org/dc/elements/1.1/title> "John phillips Blown glass" .
Inserting: <http://www3.bc.sympatico.ca/PHILLIPSHOTGLASS/GlassPage.html>
<http://purl.org/dc/elements/1.1/description> "A small display of glass by John Phillips" .
Inserting: <http://www3.bc.sympatico.ca/PHILLIPSHOTGLASS/GlassPage.html>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://directory.mozilla.org/rdf/ExternalPage> .

Asserting type rdf:Property for resource [dmoz:link]
Asserting type rdf:Property for resource [dmoz:catid]
Asserting type rdf:Property for resource [dmoz:narrow]
Asserting type [rdfs:Class] for resource [dmoz:ExternalPage]
Asserting type [rdfs:Class] for resource [dmoz:Topic]
creating property: [dmoz:narrow]
creating property: [dmoz:catid]
creating property: [dmoz:link]
Backing up class: rdfs:Resource
New property: dmoz:link for rdfs:Resource.
New property: dmoz:catid for rdfs:Resource.
New property: dmoz:narrow for rdfs:Resource.
Restoring class: rdfs:Resource
Restoring class: dctype:Collection
...
creating object: [dmoz:Topic] of rdfs:Class
creating object: [dmoz:ExternalPage] of rdfs:Class
Creating class: dmoz:ExternalPage
Creating class: dmoz:Topic
creating object: [http://www3.../GlassPage.html] of dmoz:ExternalPage
creating object: [dmoz:Top/Arts] of dmoz:Topic
creating object: [dmoz:Top] of dmoz:Topic
object: [http://www3.../GlassPage.html] predicate: dc:description value: A small display of
glass by John Phillips
object: [http://www3.../GlassPage.html] predicate: dc:title value: John phillips Blown glass
object: [dmoz:Top/Arts] predicate: dmoz:link value: [http://www3.../GlassPage.html]
object: [dmoz:Top/Arts] predicate: dc:title value: Arts
object: [dmoz:Top/Arts] predicate: dmoz:catid value: 2
object: [dmoz:Top] predicate: dmoz:narrow value: [dmoz:Top/Arts]
object: [dmoz:Top] predicate: dc:title value: Top
object: [dmoz:Top] predicate: dmoz:catid value: 1
TRUE
```