I1-D14

ERDF Implementation and Evaluation

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R/P (report and prototype)
Dissemination level:	PU (public)
Document number:	IST506779/Cottbus/I1-D14/D/PU/b1
Responsible editors:	Mircea Diaconescu
Reviewers:	Anastasia Analyti
Contributing participants:	Cottbus, Heraklion, Lisbon
Contributing workpackages:	I1
Contractual date of deliverable:	April 2008
Actual submission date:	31 March 2008

Abstract

SQL, Prolog, RDF and OWL are among the most prominent and most widely used computational logic languages. SQL, Prolog and RDF do not allow to represent negative information, only OWL does so. RDF does even not include any negation concept. While SQL and Prolog only support reasoning with closed predicates based on negation-as-failure, OWL supports reasoning with open predicates based on classical negation, only. However, in many practical application contexts, one rather needs support for reasoning with both open and closed predicates. To support this claim, we show that the well-known Web vocabulary *FOAF* includes three kinds of predicates, which we call *closed*, *open* and *partial* predicates. Therefore, reasoning with FOAF data, as a typical example of reasoning on the Web, requires a formalism that supports the distinction between open and closed predicates. We argue that *ERDF*, an extension of RDF, offers a solution to deal with this problem.

Keyword List

Rule, Reasoning, JenaRules, RDF(S), ERDF, SPARQL, FOAF.

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

ERDF Implementation and Evaluation

Gerd Wagner[1], Adrian Giurca[1], Mircea Diaconescu[1], Grigoris Antoniou[2], Carlos Viegas Damasio[3]

[1] Institute of Informatics, Brandenburg University of Technology at Cottbus,
Email: {G.Wagner, Giurca, M.Diaconescu}@tu-cottbus.de
[2]Institute of Computer Science Foundation for Research and Technology, Greece
antoniou@ics.forth.gr
[3]Universidade Nova de Lisboa, Portugal
cd@di.fct.unl.pt

31 March 2008

Abstract
SQL, Prolog, RDF and OWL are among the most prominent and most widely used computational logic languages. SQL, Prolog and RDF do not allow to represent negative information, only OWL does so. RDF does even not include any negation concept. While SQL and Prolog only support reasoning with closed predicates based on negation-as-failure, OWL supports reasoning with open predicates based on classical negation, only. However, in many practical application contexts, one rather needs support for reasoning with both open and closed predicates. To support this claim, we show that the well-known Web vocabulary *FOAF* includes three kinds of predicates, which we call *closed*, *open* and *partial* predicates. Therefore, reasoning with FOAF data, as a typical example of reasoning on the Web, requires a formalism that supports the distinction between open and closed predicates. We argue that *ERDF*, an extension of RDF, offers a solution to deal with this problem.

Keyword List
Rule, Reasoning, JenaRules, RDF(S), ERDF, SPARQL, FOAF.

Contents

1 Introduction

1.1 Open and Closed Predicates

In many information management scenarios, we deal with predicates for which we assume that we have their complete extension recorded (e.g. in a database). For such *closed* predicates, we may use the computational mechanism of *negation-as-failure (NAF)* in order to infer negative conclusions based on the explicit absence (or non-inferability) of an information item. In other words, not for *open* but only for *closed* predicates, NAF is equal to standard negation.

The issue of reasoning with closed predicates and NAF has been researched in the field of Artificial Intelligence back in the 1980's, as a form of NAF has been implemented at that time both in the database language SQL and in the logic programming language Prolog. The resulting theories and formalisms, including the famous 'Closed-World Assumption', have considered NAF to be the negation concept of choice in computational logic systems, and have downplayed the significance of 'open-world' reasoning with classical negation. 20 years later, however, a computational logic concept of classical negation has been chosen and implemented in a prominent computational logic formalism, viz the Web ontology language OWL [10]. While SQL and Prolog have a nonmonotonic computational logic semantics and support only closed predicates, OWL is based on a computational fragment of classical logic and therefore supports only open predicates. However, in many practical application contexts, one rather needs support for reasoning with both open and closed predicates.

1.2 Total and Partial Predicates

In fact, in addition to the distinction between *open* and *closed* predicates, it is useful to make another distinction between *total* and *partial* predicates. All these distinctions are related to the semantics of negative information and negation. The distinction between total and partial predicates is supported by *partial logic* (see [9]), which comes in different versions (with either 3 or 4 truth values) and can be viewed as a refinement of classical logic allowing both truth value gaps and truth value clashes. The law of the excluded middle only holds for total, but not for partial predicates. Both closed and open predicates are total. Consequently we obtain three kinds of predicates, as described in the following table:

	NAF=NEG	LEM
closed	yes	yes
open	no	yes
partial	no	no

The symbolic equation *NAF=NEG* denotes the condition that negation-as-failure and standard negation collapse, i.e. that both connectives are logically equivalent.

1.3 Three Kinds of Predicates in FOAF

A well-known example of a Web vocabulary is FOAF, the *Friends of a Friend* vocabulary [5], which is essentially expressed in RDFS (with a few additional constructs borrowed from OWL), and which has the purpose to create a Web of machine-readable information describing people, the links between them and the things they create and do.

Figure 1 defines the fragment of the FOAF vocabulary we are using in the test case.

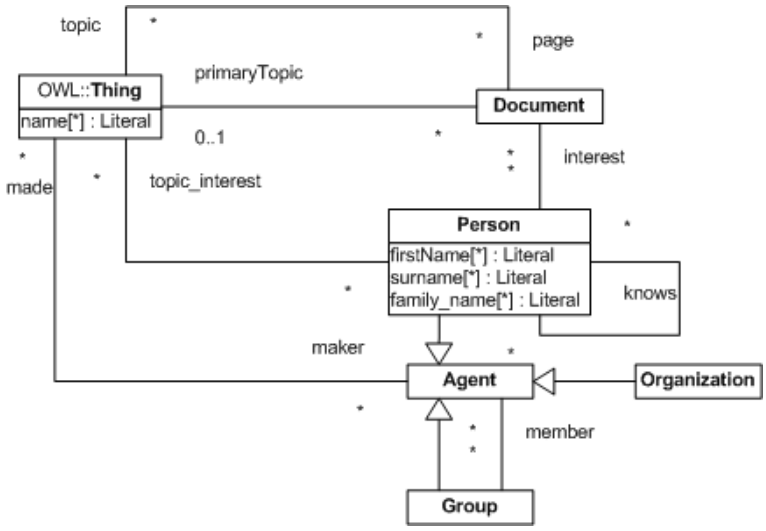

Figure 1: FOAF Vocabulary

As examples of closed, open and partial predicates included in FOAF we consider the properties `foaf:member`, `foaf:knows` and `foaf:topic_interest`. Of course, one could simply stipulate that these predicates have a standard classical logic semantics. But we argue that their intended meaning in natural language implies that they are better treated as closed, open, respectively partial predicates according to partial logic.

When a `foaf:Group` is defined, we may assume that such a definition is not made in an uncontrolled distributed manner, but rather in a controlled way where one specific person (or agent) has the authority to define the group, typically in the context of an organization that empowers the agent to do so. In this case, it is natural to consider the definition of the group membership to be a complete specification, and, consequently, to consider the `foaf:member` property to be a closed predicate. For the following example,

```
<foaf:Group rdf:ID="http://tu-cottbus.de/lit/erdf-team">
  <foaf:name>BTU Cottbus ERDF Team</foaf:name>
  <foaf:member rdf:resource="#Gerd"/>
  <foaf:member rdf:resource="#Adrian"/>
  <foaf:member rdf:resource="#Mircea"/>
</foaf:Group>
<foaf:Person rdf:ID="Gerd">
<foaf:Person rdf:ID="Adrian">
<foaf:Person rdf:ID="Mircea">
```

this would mean that we can draw the (negative) conclusion that

> Grigoris is not a member of the BTU Cottbus ERDF Team

based on the absence of a fact statement that "Grigoris is a member of the BTU Cottbus ERDF Team".

In the case of the property `foaf:knows`, however, we could argue that the standard RDF and OWL treatment of classes and properties as open predicates is adequate, since one does normally not make a complete set of statements about all persons one knows in a FOAF file. Consequently, the absence of a fact statement that "Grigoris knows Gerd" does not justify to draw the negative conclusion that "Grigoris does not know Gerd".

Both `foaf:member` and `foaf:knows` can be considered as *total* predicates that are subject to the *law of the excluded middle*, implying that the following disjunctive statements hold:

> Either Grigoris is a member of the BTU Cottbus ERDF Team or Grigoris is not a member of the BTU Cottbus ERDF Team.
> Either Grigoris knows Gerd or Grigoris does not know Gerd.

In the case of the property `foaf:topic_interest`, the situation is different. First, notice that while in the previous cases of `foaf:member` and `foaf:knows` there is no need to represent negative fact statements, we would like to be able to express both topics in which we are interested and topics in which we are definitely not interested (and would therefore prefer not to receive any news messages related to them). For instance, we may want to express the negative triple "Gerd is definitely not interested in the topic 'motor sports'".

Therefore, we should declare `foaf:topic_interest` to be a *partial* property, which means (1) that we want to be able to represent negative fact statements along with positive fact statements involving this predicate and (2) that the *law of the excluded middle* does not hold for it: it is not the case that for any topic x,

> Gerd is interested in the topic x or Gerd is (definitely) not interested in the topic x.

There may be topics, for which it is undetermined whether Gerd is interested in them or not.

1.4 Extended RDF

Since RDF(S) (see [6, 4, 7]) does not allow to represent negative information and does not support any negation concept, we need to extend it for turning it into a suitable reasoning formalism for FOAF and similar Web vocabularies.

In [12], it was argued that a database, as a knowledge representation system, needs two kinds of negation, namely *weak negation* for expressing *negation-as-failure* (or *non-truth*), and *strong negation* for expressing *explicit negative information* or *falsity*, to be able to deal with partial information. In [13], this point was also made for the Semantic Web as a framework for knowledge representation in general, and in [1, 2] for the Semantic Web language RDF with a proposal how to extend RDF for accommodating the two negations of partial logic as well as derivation rules. The extended language, called *Extended RDF*, or in short *ERDF*, has a model-theoretic semantics that is based on partial logic [9].

1.5 Plan of the Document

While the theoretical foundation of ERDF has been presented in [1, 2], the novel contributions of this deliverable are

1. an exposition and discussion of the RDF-style syntax of ERDF, and

2. a presentation of two case studies. The first, shows how a practical Web vocabulary (FOAF) would benefit from the extended logical features offered by ERDF (the support of two kinds of negation and three kinds of predicates).

In addition, we also discuss our first version of an ERDF tool set, including an inference engine.

2 The ERDF Abstract Syntax

This section describes the abstract syntax of ERDF in terms of a MOF/UML metamodel that is aligned with the RDF metamodel of OMG's *Ontology Definition Metamodel (ODM)* [8].

2.1 The ERDF-Vocabulary

ERDF allows to designate properties and classes that are completely represented in a knowledge base – they are called *closed*. The classification if a predicate is closed or not is up to the owner of the knowledge base: the owner must know for which predicates there is complete information and for which there is not. ERDF adds the following classes to the RDFS vocabulary:

- `erdf:Property` denotes the class of all properties being binary predicates that may have truth-value gaps and truth-value clashes.
- `erdf:OpenProperty`, which is equivalent to `rdf:Property`, denotes the class of all open properties being binary predicates without truth-value gaps and without truth-value clashes.
- `erdf:ClosedProperty` denotes the class of all closed properties. An ERDF ground triple with a closed property is false (that is, its strong negation can be inferred), if it cannot be inferred from the closure context of the property.
- `erdf:Class` denotes the class of all classes being unary predicates that may have truth-value gaps and truth-value clashes.
- `erdf:OpenClass`, which is equivalent to `rdfs:Class`, denotes the class of all open classes being unary predicates without truth-value gaps and without truth-value clashes.
- `erdf:ClosedClass` denotes the class of all closed classes. An ERDF ground triple formed with the rdf:type property and with a closed class as the third triple component is false (that is, its strong negation can be inferred), if it cannot be inferred from the closure context of the class.

These classes form a concept hierarchy as depicted in Figure 2. Each closed predicate comes with a *closure context*, which is the merge of a set of (E)RDF files/graphs referenced by the `erdf:closureContext` attribute. By default, if no value for `erdf:closureContext` is provided, the closure context is given by the current file.

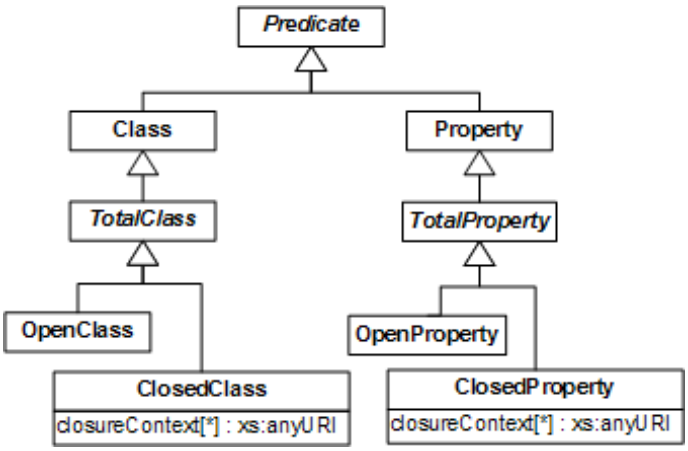

Figure 2: The ERDF vocabulary as an extension of the RDFS vocabulary

2.2 ERDF Descriptions and Atoms

ERDF descriptions, as depicted in the metamodel diagram in Figure 3, extend RDF descriptions by

1. adding to RDF property-value slots an optional attribute `erdf:negationMode` that allows to specify two kinds of negation (`Naf` for *negation-as-failure* and `Sneg` for *strong negation*);

2. allowing not only for data literals, URI references and blank node identifiers as subject and 'object' arguments (called `subjectExpr and valueExpr in Figure 3), but also for variables.`

An ERDF description consists of the following components:

- One *subject expression*, denoted by the `subjectExpr` property in the metamodel diagram, being an *ERDF term*, that is a `URIReference`, a `Variable`, an `ExistentialVariable` (blank node identifier) or `rdfs:Literal` (see Figure 4 for the definition of ERDF term).

- A non-empty set of *slots* being property-value pairs consisting of a URI reference denoting a property and an ERDF term as the value expression.

Obviously, descriptions with just one slot correspond to the usual concept of an atomic statement (or triple), while descriptions with multiple slots correspond to conjunctions of such statements. However, as can be seen in Figure 3, all descriptions are considered as ***ERDF atoms***, which in addition subsume *datatype predicate* atoms (notice that datatype predicates are often also called 'built-ins').

ERDF fact statements are variable-free ERDF descriptions such that no slot has a negation mode other than `None` or `Sneg`. That is, only strong negation may occur in fact statements (in the case of negative information).

ERDF descriptions with variables correspond to conjunctive query formulas that can be used as rule conditions.

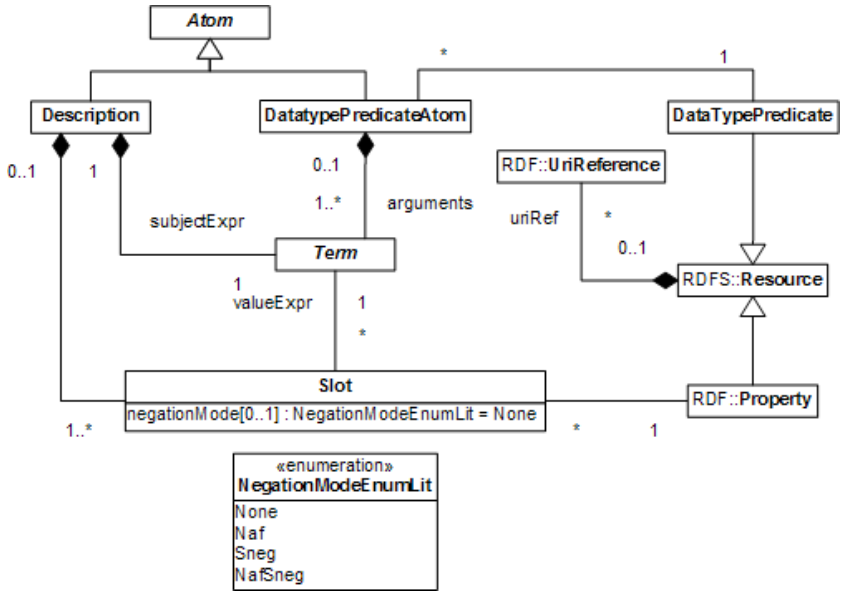

Figure 3: ERDF Descriptions

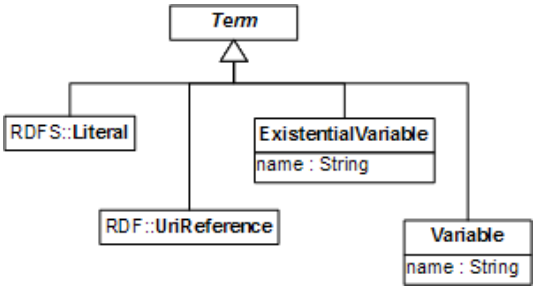

Figure 4: ERDF Terms

2.3 ERDF Rules

The abstract syntax of ERDF rules is defined in the metamodel diagram in Figure 5. ERDF rules are derivation rules of the form $D \leftarrow A_1, \ldots, A_n$. where D is an ERDF description with only `None` or `Sneg` as slot negation modes and $A_1, \ldots, A_n$ are *ERDF atoms*, that is, descriptions or datatype predicate atoms.

3 Concrete Syntaxes for ERDF

3.1 XML Syntax

Our approach is to follow the RDF/XML syntax as much as possible and derive an RDF-style syntax for ERDF atomic formulas ('triple patterns') from the abstract syntax metamodel

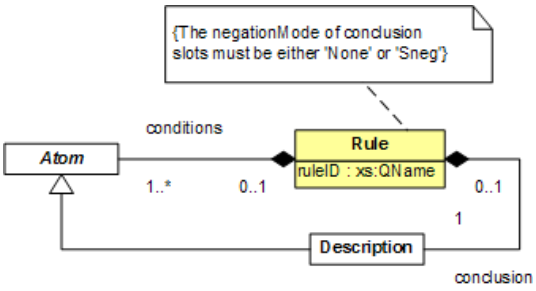

Figure 5: ERDF Rule

presented above.

3.1.1 Expressing a Vocabulary in ERDF

Using the ERDF predicate categories defined in section 2.1, we can refine the FOAF vocabulary definition of `foaf:member`, `foaf:knows` and `foaf:topic_interest` as follows:

```
<erdf:OpenProperty rdf:about="http://xmlns.com/foaf/0.1/knows">
  <rdfs:domain rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
  <rdfs:range rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
</erdf:OpenProperty>

<erdf:PartialProperty rdf:about="http://xmlns.com/foaf/0.1/topic_interest">
  <rdfs:domain rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
  <rdfs:range rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
</erdf:PartialProperty>

<erdf:ClosedProperty rdf:about="http://xmlns.com/foaf/0.1/member">
  <rdfs:domain rdf:resource="http://xmlns.com/foaf/0.1/Group"/>
  <rdfs:range rdf:resource="http://xmlns.com/foaf/0.1/Agent"/>
</erdf:ClosedProperty>
```

We identify `erdf:OpenProperty` with `rdf:Property` and `erdf:OpenClass` with `rdfs:Class`. Thus, by default, all RDF predicates are considered to be open.

3.1.2 Expressing ERDF Terms

ERDF terms are URI references, blank node identifiers, variables or data literals. They are expressed in two ways, depending on their occurrence as *subject expressions* or as *value expressions*.

Terms as subject expressions are values of the `erdf:about` attribute, which may be URI references, blank node identifiers or variables using the SPARQL syntax for blank node identifiers and variables.

Terms as value expressions are expressed either with the help of one of the attributes `rdf:resource`, `rdf:nodeID` or `erdf:variable`, or as the text content of the property-value slot element in the case of a data literal.

3.1.3 Descriptions and Datatype Predicate Atoms

ERDF descriptions are encoded by means of the `erdf:Description` element. Each description contains a non-empty list of (possibly negated) property-value slots.

Example 1 *Gerd knows Adrian, has some topic interest, but is not interested in the topic 'motor sports'*

```
<erdf:Description erdf:about="#Gerd">
  <foaf:knows rdf:resource="#Adrian"/>
  <foaf:topic_interest rdf:nodeID="x"/>
  <foaf:topic_interest erdf:negationMode="Sneg"
                       rdf:resource="urn:topics:motor_sports"/>
</erdf:Description>
```

Datatype predicate atoms are n-ary logical atoms. The value of `erdf:arguments` property represent an ordered list of arguments. The `erdf:predicate` XML attribute encodes the URI reference to the predicate.

Example 2

```
<erdf:DatatypePredicateAtom erdf:predicate="swrlb:add">
  <erdf:Variable>?sum</erdf:Variable>
  <rdfs:Literal rdf:datatype="xs:int">40</rdfs:Literal>
  <rdfs:Literal rdf:datatype="xs:int">20</rdfs:Literal>
</erdf:DatatypePredicateAtom>
```

3.1.4 Rules and Rulesets

We propose two syntaxes for ERDF rules: a more concise non-XML syntax based on SPARQL triple patterns where universally quantified variables are expressed by prefixing a name with '?', and an XML-based syntax, which will be useful for rule transformations and interchange.

In order to express ERDF rules in XML, we use the rule markup language R2ML [14]. It may be an option later to use the W3C rule interchange format, if the W3C RIF working group will deliver a usable recommendation in 2008.

3.1.5 ERDF Triple Pattern Syntax

Actually, JenaRules abstract syntax is expressed as following:

```
Rule      :=   bare-rule .
          or   [ bare-rule ]
          or   [ ruleName : bare-rule ]

bare-rule :=   term, ... term -> hterm, ... hterm    // forward rule
          or   term, ... term <- term, ... term      // backward rule

hterm     :=   term
          or   [ bare-rule ]

term      :=   (node, node, node)           // triple pattern
          or   (node, node, functor)        // extended triple pattern
          or   builtin(node, ... node)      // invoke procedural primitive
```

```
functor   :=   functorName(node, ... node)  // structured literal

node      :=   uri-ref                       // e.g. http://foo.com/eg
          or   prefix:localname             // e.g. rdf:type
          or   <uri-ref>                       // e.g. <myscheme:myuri>
          or   ?varname                     // variable
          or   'a literal'                  // a plain string literal
          or   'lex'^^typeURI               // a typed literal, xsd:* type names supported
          or   number                       // e.g. 42 or 25.5
```

The main entities for rules are triples. The triple is presented in two forms:

- expressed by three nodes, `(subject, predicate, object)`, i.e. `(?x foaf:likes ex:John)`;
- expressed by two nodes, `subject` and `predicate` and a syntactical expression for a functor, i.e. `(?x ex:discount calc(?totalValue))`;

3.1.6 ERDF Turtle syntax

The new structure will add a new type of triple, namely strong negated triple, expressing a negative triple. The syntax of this new triple is: `term := (node, -node, node)`. The "-" symbol in front of the predicate (the second position in triple) express that the triple is strong negated. For properties, this means that we have this triple in the negative extension of the predicate. For classes this is defined by using `-rdf:type` property.

Since in ERDF we have two forms of negation, we have to define also negation as failure. For JenaRules syntax this is defined under the form of a builtin. The builtin expressing negation as failure is named `naf`, and it can have two forms, with two or three arguments:

- in the form with three arguments (e.g. `naf(?x ex:likes tcw:Merlot)` or using strong negation `naf(?x -ex:likes tcw:Retsina)`) it looks in the positive or negative extension of the predicate (depending if the argument triple it is or not negated) and return true if the triple is not found, and false otherwise.
- in the form with two arguments (e.g. `naf(?x ex:likes)` or `naf(?x -ex:likes)`) it looks in the extension (positive or negative, depending if the argument triple is or not negated) and looks for every triple which has the value of `?x` if x is bounded or looks for every triple if `?x` is not bounded. In this form, the object (the third position in triple) is ignored, meaning that it can be anything. This is equivalent with `naf(?s ex:p ?o)` where `?o` is an unbounded variable.

4 The ERDF Application Programming Interface

This section describe how we extend Jena API in order to express and reason with rules over ERDF facts base. We also explain the main structure of the ERDF API v0.1.

4.1 Engine architecture

This section describe changes and adds for JenaAPI in order to support reasoning over ERDF knowledge bases.

4.1.1 Jena Architecture Overview

The Jena2 inference subsystem is designed to allow a range of inference engines or reasoners to be plugged into Jena. Such engines are used to derive additional RDF assertions which are entailed from some base RDF together with any optional ontology information and the axioms and rules associated with the reasoner. The primary use of this mechanism is to support the use of languages such as RDFS and OWL which allow additional facts to be inferred from instance data and class descriptions. A general structure of the inference API is expressed in the Fig. 6.

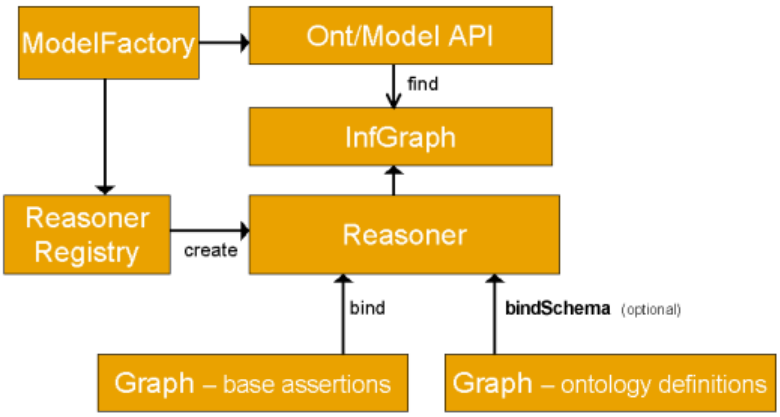

Figure 6: Jena Inference API

The access to the inference machinery is made by using `ModelFactory` to associate a data set with some reasoner to create a new `Model`. Queries to the created model will return not only those statements that were present in the original data but also additional statements than can be derived from the data using the rules or other inference mechanisms implemented by the reasoner.

Actually, in the Jena distribution are included a number of predefined reasoners:

- `Transitive reasoner` - provides support for storing and traversing class and property lattices. This implements just the transitive and symmetric properties of `rdfs:subPropertyOf` and `rdfs:subClassOf`.
- `RDFS rule reasoner` - implements a configurable subset of the RDFS entailments.
- `OWL, OWL Mini, OWL Micro Reasoners` - a set of useful but incomplete implementation of the OWL/Lite subset of the OWL/Full language.
- `DAML micro reasoner` - used internally to enable the legacy DAML API to provide minimal (RDFS scale) inferencing.
- `Generic rule reasoner` - a rule based reasoner that supports user defined rules. Forward chaining, tabled backward chaining and hybrid execution strategies are supported.

4.1.2 Extending Jena API to support reasoning over ERDF

Actually, JenaAPI uses concepts of `Rule`, `TriplePattern`, `Node` and `Builtin` (see Fig. 7).

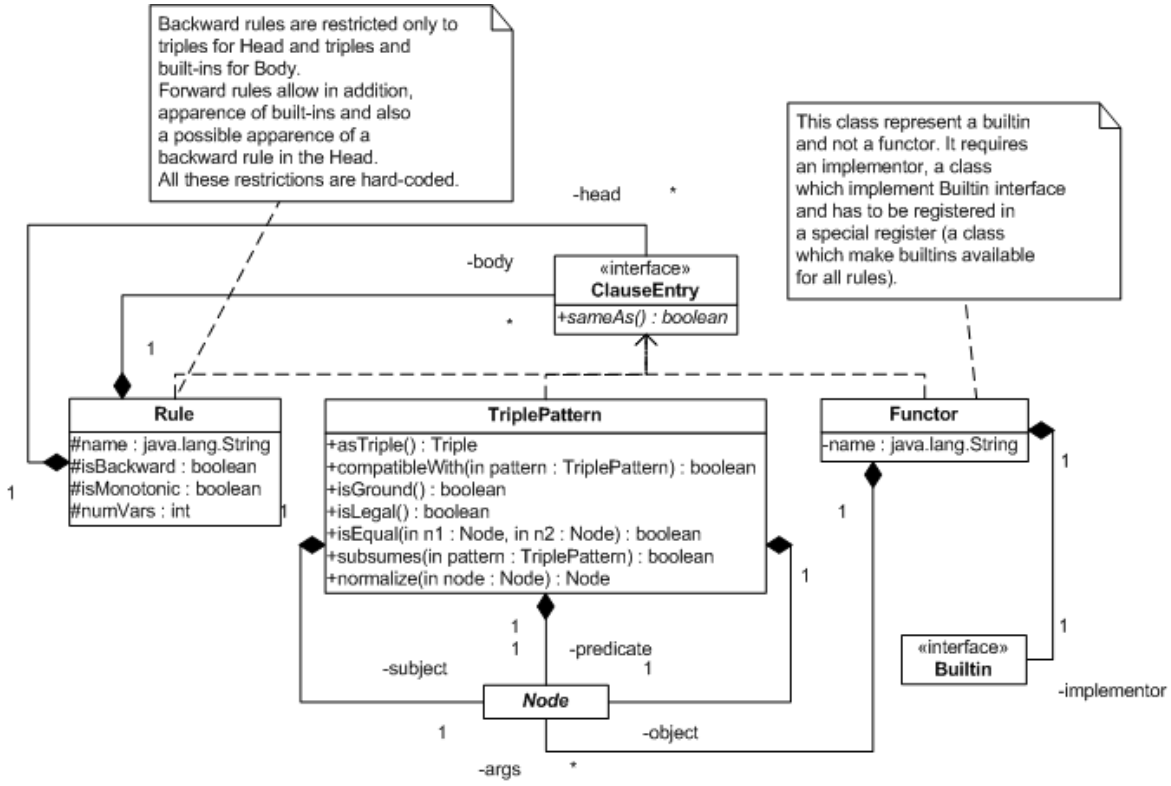

Figure 7: Actual structure of a Jena Rule

In order to reason over ERDF facts, we have to extend actual structure with a new class, named `NegTriplePattern` which represent strong negated triples in a rule. Also, we add `PosTriplePattern` class, representing a positive triple pattern. Both classes extend `TriplePattern` class and implement `ClauseEntry` interface in order to maintain compatibility with actual JenaAPI structure (see Fig. 8).

`NegTriplePattern` class will be instantiated every time in a rule when we have a strong negated triple. This is also used in the reasoning process, since it make difference between the two types of triples: positive and negative. Every triple pattern contains three nodes: `subject`, `predicate` and `object`. There are multiple types of nodes defined in JenaRules (see Fig. 9).

Also in a rule, we can use built-ins. Built-ins parameters are nodes. We implement a special builtin, `naf`, representing negation-as-failure. The basic structure of built-ins in JenaAPI is depicted in Fig. 10.

The next step is to extend Jena with a new class, `NegTriple`, which express a negative ground triple. These triples are obtained from initial facts, and from deductions. This class has to extend `Triple` class from Jena in order to maintain compatibility with the actual JenaAPI structure (see Fig. 11). Instances of `NegTriple` and `Triple` are added to a `Graph`.

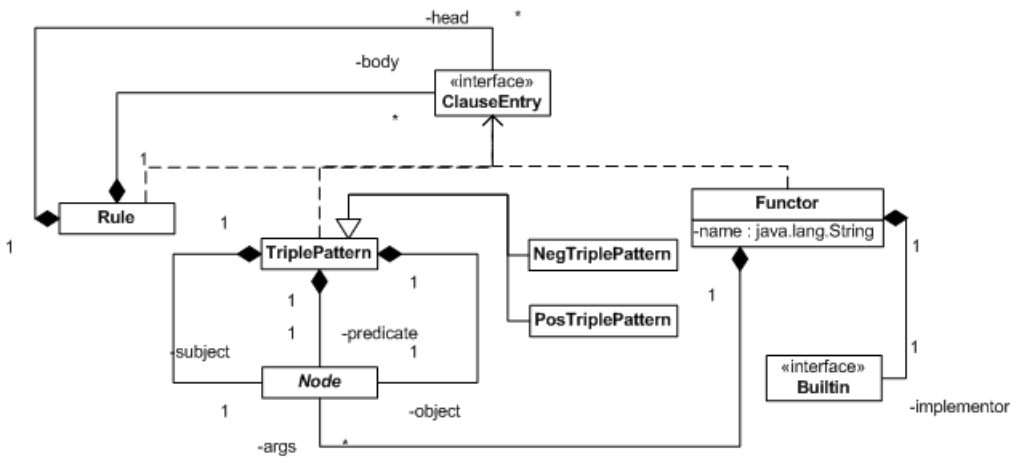

Figure 8: Extended structure of a Jena Rule

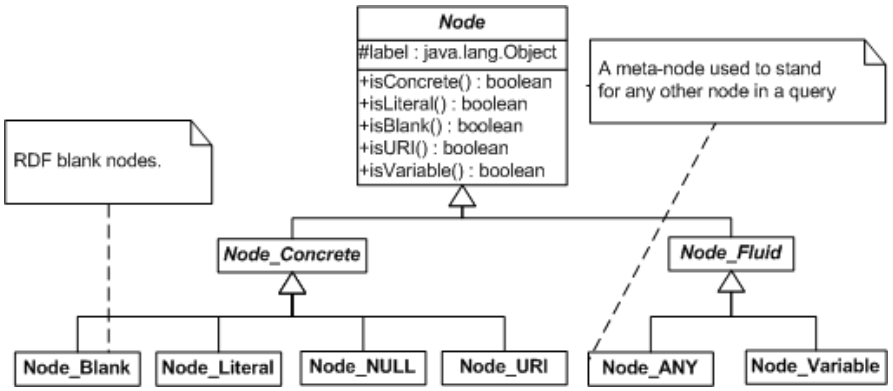

Figure 9: Node types

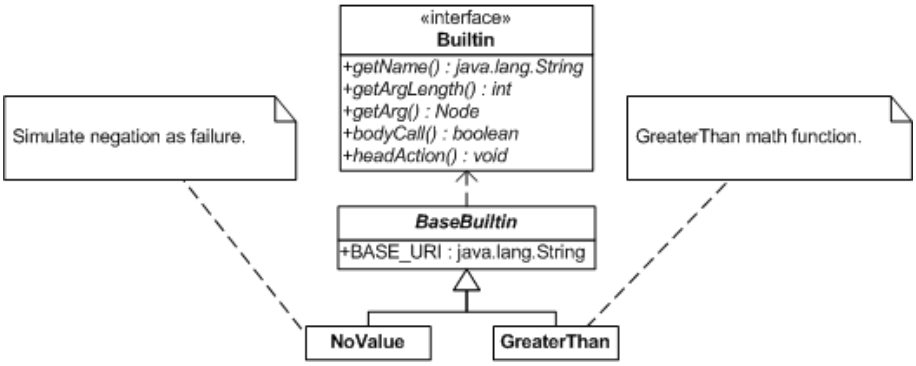

Figure 10: Jena built-ins

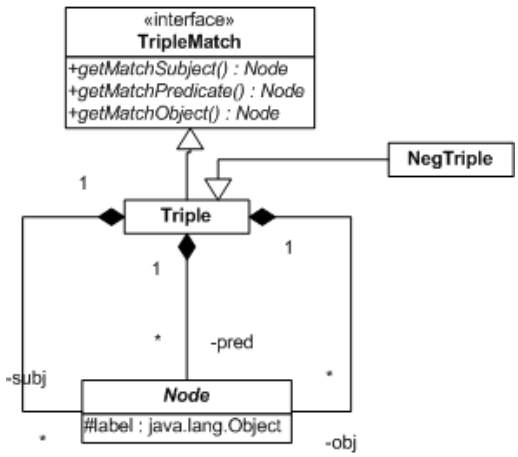

Figure 11: Extending Jena Triple with NegTriple

Before creating an inference model, the given original data (e.g. from a ERDF file), is need to be parsed by an extended RDF parser. The extended RDF parser is able to detect and handle both positive and negative ERDF triples. Finally the data will be stored into a graph.

The next step is to create and configure the `ERDFReasoner`. The `ERDFReasoner` is equipped with a `RuleStore` in which the rules are stored. First, rules (e.g. from a given rule file), have to be parsed. This job is done by an extended rule parser which is be able to handle negation information in rules. After parsing, rules are stored in the `RuleStore`.

Now we are able to create an inference model. When creating a new inference model, with the help of the `ModelFactory`, the graph in which the original data is stored, has to be attached (or bind) to the `ERDFReasoner`.

After the initialization of the inference model we are able to query the InfModel. Finally such a query is handled by the `ERDFRuleEngine` which uses `ERDFInterpreter` to generate the results for each query. In order o obtain the results for a given query, the `ERDFRuleEngine` access both the data from the `Graph` and the rules from the `RuleStore`. The final result is then returned to the `InfModel`. Fig. 12 express the flux for inference with rules over ERDF facts.

The actual structure of ERDF Reasoner based on the Jena Reasoner structure, is depicted in Fig. 13.

4.2 ERDF Inference

Since now we have to deal with different types of classes and properties, such as `Total`, `Partial`, `Open` or `Closed`, we have to express the possibility of of calculate the transitive closure of ERDF knowledge base based on the transitive closure of RDFS. Jena API use internally a set of axioms and facts which are capable to calculate the transitive closure of RDF(S) knowledge base. For example, we have an axiom which say that the range of the `rdf:type` property is `rdfs:Class`, and one which specify that every resource is of type `rdfs:Class`.

```
-> (rdf:type      rdfs:range rdfs:Class).
-> (rdfs:Resource  rdf:type  rdfs:Class).
```

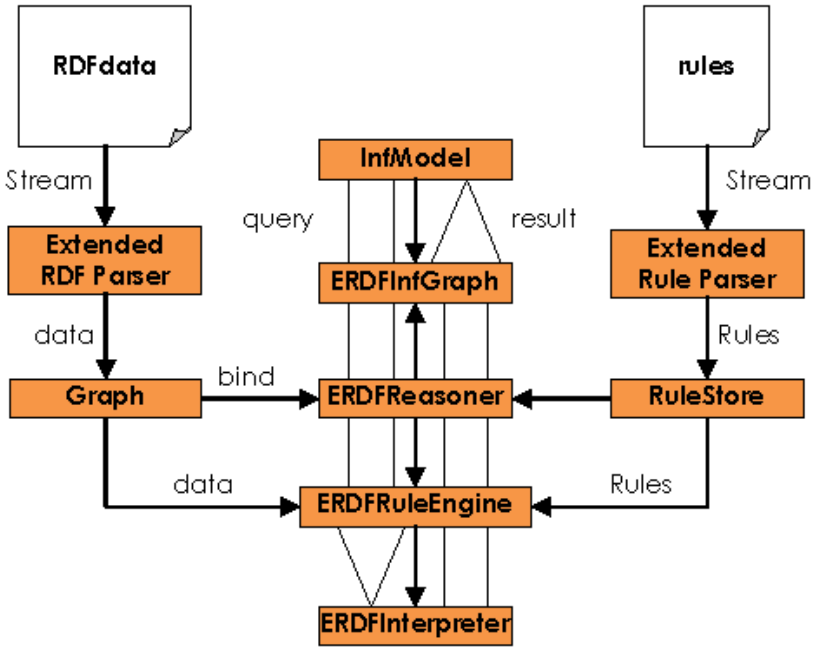

Figure 12: Reasoning with ERDFReasoner

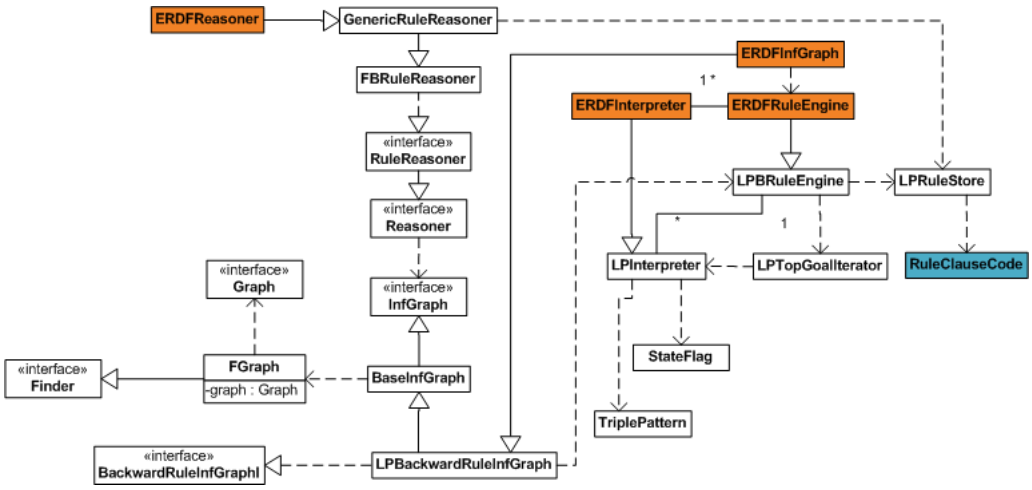

Figure 13: ERDF Reasoner structure

We have also a set of rules defining how the RDF(S) transitive closure is calculated. For example, we have a rule for `rdfs:subClassOf` property:

```
[(?a rdfs:subClassOf ?b), (?b rdfs:subClassOf ?c) -> (?a rdfs:subClassOf ?c)]
```

Since we work now in ERDF, the superclass of all classes is `erdf:Class` and the superclass of all properties is `erdf:Propery`.

The full set of axioms and rules is defined in a rule file, which is loaded by default when the RDFS reasoner is created. The content of the modified set of axioms and rules is presented below:

```
// axioms
-> (rdf:type      rdfs:range erdf:Class).
-> (rdfs:Resource  rdf:type  erdf:Class).
-> (rdfs:Literal   rdf:type  erdf:Class).
-> (rdf:Statement  rdf:type  erdf:Class).
-> (rdf:nil        rdf:type  rdf:List).
-> (rdf:subject    rdf:type  erdf:Property).
-> (rdf:object     rdf:type  erdf:Property).
-> (rdf:predicate  rdf:type  erdf:Property).
-> (rdf:first      rdf:type  erdf:Property).
-> (rdf:rest       rdf:type  erdf:Property).

-> (rdfs:subPropertyOf rdfs:domain erdf:Property).
-> (rdfs:subClassOf rdfs:domain erdf:Class).
-> (rdfs:domain rdfs:domain erdf:Property).
-> (rdfs:range rdfs:domain erdf:Property).
-> (rdf:subject rdfs:domain rdf:Statement).
-> (rdf:predicate rdfs:domain rdf:Statement).
-> (rdf:object rdfs:domain rdf:Statement).
-> (rdf:first rdfs:domain rdf:List).
-> (rdf:rest rdfs:domain rdf:List).

-> (rdfs:subPropertyOf rdfs:range erdf:Property).
-> (rdfs:subClassOf rdfs:range erdf:Class).
-> (rdfs:domain rdfs:range erdf:Class).
-> (rdfs:range rdfs:range erdf:Class).
-> (rdf:type rdfs:range erdf:Class).
-> (rdfs:comment rdfs:range rdfs:Literal).
-> (rdfs:label rdfs:range rdfs:Literal).
-> (rdf:rest rdfs:range rdf:List).

-> (rdf:Alt rdfs:subClassOf rdfs:Container).
-> (rdf:Bag rdfs:subClassOf rdfs:Container).
-> (rdf:Seq rdfs:subClassOf rdfs:Container).
-> (rdfs:ContainerMembershipProperty rdfs:subClassOf erdf:Property).

-> (rdfs:isDefinedBy rdfs:subPropertyOf rdfs:seeAlso).

-> (rdf:XMLLiteral rdf:type rdfs:Datatype).
-> (rdfs:Datatype rdfs:subClassOf erdf:Class).

//RDFS Closure Rules
[rdf1and4: (?x ?p ?y)
           ->
           (?p rdf:type erdf:Property)
           (?x rdf:type rdfs:Resource)
           (?y rdf:type rdfs:Resource)]
[rdfs7b: (?a rdf:type erdf:Class) -> (?a rdfs:subClassOf rdfs:Resource)]

[rdfs2:  (?x ?p ?y), (?p rdfs:domain ?c) -> (?x rdf:type ?c)]
```

```
[rdfs3:  (?x ?p ?y), (?p rdfs:range ?c) -> (?y rdf:type ?c)]
[rdfs5a: (?a rdfs:subPropertyOf ?b), (?b rdfs:subPropertyOf ?c)
         ->
         (?a rdfs:subPropertyOf ?c)]
[rdfs5b: (?a rdf:type erdf:Property) -> (?a rdfs:subPropertyOf ?a)]
[rdfs6:  (?a ?p ?b), (?p rdfs:subPropertyOf ?q) -> (?a ?q ?b)]
[rdfs7:  (?a rdf:type erdf:Class) -> (?a rdfs:subClassOf ?a)]
[rdfs8:  (?a rdfs:subClassOf ?b), (?b rdfs:subClassOf ?c)
         ->
         (?a rdfs:subClassOf ?c)]
[rdfs9:  (?x rdfs:subClassOf ?y) (?a rdf:type ?x) -> (?a rdf:type ?y)]
[rdfs10: (?x rdf:type rdfs:ContainerMembershipProperty)
         ->
         (?x rdfs:subPropertyOf rdfs:member)]
```

Using the same method, we define a set of axioms and rules to calculate the transitive closure for ERDF. This, implies also the usage of modified RDFS axioms. In addition, a set of set of axioms is defined for ERDF:

```
-> (erdf:TotalClass rdfs:subClassOf erdf:Class)
-> (erdf:PartialClass rdfs:subClassOf erdf:Class)

-> (erdf:ClosedClass rdfs:subClassOf erdf:TotalClass)
-> (erdf:OpenClass rdfs:subClassOf erdf:TotalClass)

-> (erdf:TotalProperty rdfs:subClassOf erdf:Property)
-> (erdf:PartialProperty rdfs:subClassOf erdf:Property)

-> (erdf:ClosedProperty rdfs:subClassOf erdf:TotalProperty)
-> (erdf:OpenProperty rdfs:subClassOf erdf:TotalProperty)

-> (erdf:OpenClass rdfs:subClassOf rdfs:Class)
-> (rdfs:Class rdfs:subClassOf erdf:OpenClass)

-> (erdf:OpenProperty rdfs:subClassOf rdf:Property)
-> (rdf:Property rdfs:subClassOf erdf:OpenProperty)
```

Those axioms establish the `rdfs:subClassOf` relationship between the ERDF newly added classes, and the similarities between RDF(S) and ERDF: `rdfs:Class` is the same with `erdf:OpenClass` and `rdf:Property` is identified by `erdf:OpenPropery`.

Since we have to deal with properties which are closed, a set of rules to calculate the transitive closure of ERDF has to be defined. This has to take into consideration that, if a set of facts regarding a specific property, which is closed, does not represent a negated triple for a specific resource from the range of that property, then it has to represent mandatory an positive triple (also the reverse situation has to be accomplished).

The following rules define the transitive closure for closed properties:

```
[close1: (?s -?p ?o)
         <-
         (?p rdf:type erdf:ClosedProperty)
         (?p rdf:range ?r)(?p rdf:domain ?d)
         (?s rdf:type ?d)(?o rdf:type ?r)
         naf(?s ?p ?o)]
```

```
[close2: (?s ?p ?o)
         <-
         (?p rdf:type erdf:ClosedProperty)
         (?p rdf:range ?r)(?p rdf:domain ?d)
         (?s rdf:type ?d)(?o rdf:type ?r)
         naf(?s -?p ?o)]
```

The set of axioms and rules which are presented above are used by the ERDFReasoner to calculate the transitive closure of ERDF, after the RDFS transitive closure is calculated.

5 Cases Study

5.1 Building FOAF-Based Working Groups

This part presents a scenario involving FOAF data and ERDF rules. We refer to a scenario where an organizing committee has to form working groups with people from different communities. The assumption is that every member has a FOAF file describing her topic interests and contacts. All these files are available to the organizing committee. The task of the organizers is to try to create working groups for different areas of topic interests, by recommending to members to be in groups which have the same interest domain. The goal is also to put together participants which do not know each other, and which do not have contradictory topic interests. By a contradictory topic interest we mean that a topic interest of a member is negated in the FOAF file of the other member.

The information about the contacts of a participant is represented using the `foaf:knows` property.

Consider, for example, the following facts collected from the FOAF files of four meeting participants:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:erdf="http://www.informatik.tu-cottbus.de/IT/erdf#">

  <erdf:Description erdf:about="#Gerd">
    <rdf:type rdf:resource="foaf:Person"/>
    <foaf:topic_interest rdf:resource="urn:topics:RDF"/>
    <foaf:topic_interest rdf:resource="urn:topics:AgentBasedSimulation"/>
    <foaf:topic_interest erdf:negationMode="Sneg"
                         rdf:resource="urn:topics:motor_sports"/>
  </erdf:Description>

  <rdf:Description erdf:about="#Grigoris">
    <rdf:type rdf:resource="foaf:Person"/>
    <foaf:knows rdf:resource="#Gerd"/>
    <foaf:topic_interest rdf:resource="urn:topics:RDF"/>
  </rdf:Description>

  <rdf:Description erdf:about="#Pete">
    <rdf:type rdf:resource="foaf:Person"/>
    <foaf:topic_interest rdf:resource="urn:topics:RDF"/>
    <foaf:topic_interest rdf:resource="urn:topics:motor_sports"/>
  </rdf:Description>
```

```
  <rdf:Description erdf:about="#Tom">
    <rdf:type rdf:resource="foaf:Person"/>
    <foaf:knows rdf:resource="#Pete"/>
    <foaf:topic_interest rdf:resource="urn:topics:AgentBasedSimulation"/>
  </rdf:Description>
</rdf:RDF>
```

Notice that only the first description, since it includes a negative triple, needs to be marked up as an ERDF description. For the other (positive) fact statements we simply reuse RDF.

We define a rule which can help to make suggestions (or recommendations) for grouping different participants as members of the same working group. This rule takes in consideration if X knows Y (or Y knows X) and if there are common topic interests between the two participants, and no contradictory topic interests:

> If persons X and Y do not know each other and they have at least one common topic interest and do not have any contradictory topic interest, then recommend that X and Y be in the same group.

Using Extended Jena Syntax, we can now describe the above rule under the form of three derivation rules:

```
[contradictoryInterest1: (?x conf:sharesContradictoryInterestWith ?y)
                         <-
                         (?x rdf:type foaf:Person)
                         (?y rdf:type foaf:Person)
                         (?x foaf:topic_interest ?t)
                         (?y -foaf:topic_interest ?t)]

[contradictoryInterest2: (?x conf:sharesContradictoryInterestWith ?y)
                         <-
                         (?x rdf:type foaf:Person)
                         (?y rdf:type foaf:Person)
                         (?x -foaf:topic_interest ?t)
                         (?y foaf:topic_interest ?t)]

[sameGroup: (?x conf:sameGroupAs ?y)
            <-
            (?x rdf:type foaf:Person)
            (?y rdf:type foaf:Person)
            naf(?x foaf:knows ?y)
            naf(?y foaf:knows ?x)
            naf(?x conf:sharesContradictoryInterestWith ?y)]
```

The property `conf:sameGroupAs` suggests that the two persons may be in the same group. This property can be used in order to define other rules which can create `foaf:Group` class instances and add `foaf:member`'s to the group. Then, we can query such a fact base in order to see all members of a group.

In order to express rules in XML Syntax, we can use R2ML Markup. The above `sameGroupAs` rule is expressed in R2ML XML syntax in the following example:

```
<r2ml:DerivationRule r2ml:ruleID="sameGroupAs">
  <r2ml:conditions>
```

```
    <erdf:Description erdf:about="?x">
      <rdf:type rdf:resource="foaf:Person"/>
      <foaf:knows erdf:negationMode="naf" erdf:variable="?y"/>
      <conf:sharesContradictoryInterestWith erdf:negationMode="naf" erdf:variable="?y"/>
    </erdf:Description>
    <erdf:Description erdf:about="?y">
      <rdf:type rdf:resource="foaf:Person"/>
      <foaf:knows erdf:negationMode="naf" erdf:variable="?x"/>
    </erdf:Description>
  </r2ml:conditions>
  <r2ml:conclusion>
    <erdf:Description erdf:about="?x">
      <conf:sameGroupAs erdf:variable="?y"/>
    </erdf:Description>
  </r2ml:conclusion>
</r2ml:DerivationRule>
```

In order to express this rule, it is also needed to deduct facts regarding `conf:sharesContradictoryInterestWith`. This is expressed using another rule(s): `contradictoryInterest1`, and `contradictoryInterest2`. The R2ML XML markup for `contradictoryInterest1` rule is depicted below:

```
<r2ml:DerivationRule r2ml:ruleID="contradictoryInterest1">
  <r2ml:conditions>
    <erdf:Description erdf:about="?x">
      <rdf:type rdf:resource="foaf:Person"/>
      <foaf:topic_interest erdf:variable="?t"/>
    </erdf:Description>
    <erdf:Description erdf:about="?y">
      <rdf:type rdf:resource="foaf:Person"/>
      <foaf:topic_interest erdf:negationMode="sneg" erdf:variable="?t"/>
    </erdf:Description>
  </r2ml:conditions>
  <r2ml:conclusion>
    <erdf:Description erdf:about="?x">
      <conf:sharesContradictoryInterestWith erdf:variable="?y"/>
    </erdf:Description>
  </r2ml:conclusion>
</r2ml:DerivationRule>
```

The two rules expressed using R2ML are depicted into a compact form, meaning that we group properties of the same `subject`. For example, we say:

```
<erdf:Description erdf:about="?x">
  <rdf:type rdf:resource="foaf:Person"/>
  <foaf:topic_interest erdf:variable="?t"/>
</erdf:Description>
```

to express 'all' about '?x' subject.

5.2 Conference Dinner Wines

We propose now, another scenario which express ERDF based reasoning. The test case refers to a dinner where different guest are invited, and the organizing committee have to know which wines can be or not served at the conference dinner. In order to do that, they know already for each guest the wines which they likes and the wines which the doesn't likes.

Using ERDF XML Syntax we express the schema to represent facts for this situation:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 xmlns:erdf="http://www.informatik.tu-cottbus.de/IT/erdf#"
 xmlns:xs="http://www.w3.org/2001/XMLSchema#">

  <rdfs:Class rdf:about="http://example.com/dinner#Guest"/>
  <rdfs:Class rdf:about="http://example.com/dinner#Wine" />
  <rdfs:Class rdf:about="http://example.com/dinner#AvailableWine">
    <rdfs:subClassOf rdf:resource="http://example.com/dinner#Wine" />
  </rdfs:Class>
  <erdf:PartialClass rdf:about="http://example.com/dinner#WineForDinner">
    <rdfs:subClassOf rdf:resource="http://example.com/dinner#Wine" />
  </erdf:PartialClass>
  <rdf:Property rdf:about="http://example.com/dinner#name">
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal" />
    <rdfs:domain rdf:resource="http://example.com/dinner#Guest" />
  </rdf:Property>
  <erdf:PartialProperty rdf:about="http://example.com/dinner#likes">
    <rdfs:range rdf:resource="http://example.com/dinner#Wine" />
    <rdfs:domain rdf:resource="http://example.com/dinner#Guest" />
  </erdf:PartialProperty>
</rdf:RDF>
```

We combine RDF(S) with ERDF, since we use ERDF structures only where we need to express specific ERDF elements, otherwise we use RDF(S) elements. Below, we express some facts based on the above defined schema:

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:erdf="http://www.informatik.tu-cottbus.de/IT/erdf#"
 xmlns:tcw="http://example.com/dinner#"
 xml:base="http://example.com/dinner#">

  <rdf:Description rdf:about="http://example.com/dinner#Riesling">
    <rdf:type rdf:resource="http://example.com/dinner#AvailableWine" />
  </rdf:Description>

  <rdf:Description rdf:about="http://example.com/dinner#Retsina">
    <rdf:type rdf:resource="http://example.com/dinner#AvailableWine" />
  </rdf:Description>

  <rdf:Description rdf:about="http://example.com/dinner#Merlot">
    <rdf:type rdf:resource="http://example.com/dinner#AvailableWine" />
  </rdf:Description>

  <rdf:Description rdf:about="http://example.com/dinner#Chardonay">
    <rdf:type rdf:resource="http://example.com/dinner#AvailableWine" />
  </rdf:Description>

  <rdf:Description rdf:about="http://example.com/dinner#VinoVerde">
    <rdf:type rdf:resource="http://example.com/dinner#AvailableWine" />
  </rdf:Description>

  <rdf:Description rdf:about="http://example.com/dinner#PinotNoir">
    <rdf:type rdf:resource="http://example.com/dinner#AvailableWine" />
```

```
  </rdf:Description>

  <erdf:Description rdf:about="Gerd">
    <tcw:name>Gerd Wagner</tcw:name>
    <tcw:likes rdf:resource="http://example.com/dinner#Riesling" />
    <tcw:likes erdf:negationMode="sneg"
               rdf:resource="http://example.com/dinner#Retsina" />
    <tcw:likes erdf:negationMode="sneg"
               rdf:resource="http://example.com/dinner#Merlot" />
    <rdf:type rdf:resource="http://example.com/dinner#Guest" />
  </erdf:Description>

  <erdf:Description rdf:about="Anastasia">
    <tcw:name>Anastasia Analyti</tcw:name>
    <tcw:likes erdf:negationMode="sneg"
               rdf:resource="http://example.com/dinner#Retsina" />
    <tcw:likes rdf:resource="http://example.com/dinner#Merlot" />
    <tcw:likes rdf:resource="http://example.com/dinner#Chardonay" />
    <rdf:type rdf:resource="http://example.com/dinner#Guest" />
  </erdf:Description>

  <rdf:Description rdf:about="Carlos">
    <tcw:name>Carlos Viegas Damasio</tcw:name>
    <tcw:likes rdf:resource="http://example.com/dinner#VinoVerde" />
    <tcw:likes rdf:resource="http://example.com/dinner#Merlot" />
    <rdf:type rdf:resource="http://example.com/dinner#Guest" />
  </rdf:Description>

  <rdf:Description rdf:about="Grigoris">
    <tcw:name>Grigoris Antoniu</tcw:name>
    <tcw:likes rdf:resource="http://example.com/dinner#Merlot" />
    <tcw:likes rdf:resource="http://example.com/dinner#Chardonay" />
    <rdf:type rdf:resource="http://example.com/dinner#Guest" />
  </rdf:Description>

  <rdf:Description rdf:about="Mircea">
    <tcw:name>Mircea Diaconescu</tcw:name>
    <tcw:likes rdf:resource="http://example.com/dinner#Riesling" />
    <tcw:likes rdf:resource="http://example.com/dinner#Merlot" />
    <rdf:type rdf:resource="http://example.com/dinner#Guest" />
  </rdf:Description>

</rdf:RDF>
```

We want to see which wines are appropriate to be served at the dinner and which are not. In order to do that, we define derivation rules which derive the concepts of *WineForDinner* and *NotWineForDinner*. The rules are defined in the form of the extended JenaRules Syntax.

```
  @prefix rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
  @prefix erdf: http://www.informatik.tu-cottbus.de/IT/erdf#
  @prefix tcw: http://informatik.tu-cottbus.de/erdf/usecases/dinner#

  // wines for dinner
  [wineForDinner: (?wine rdf:type tcw:WineForDinner)
                  <-
                  (?wine rdf:type tcw:AvailableWine)
```

```
                (?guest rdf:type tcw:Guest)
                (?guest tcw:likes ?wine)]

// wines which are not for dinner
[notWineForDinner: (?wine -rdf:type tcw:WineForDinner)
                   <-
                   (?wine rdf:type tcw:AvailableWine)
                   (?guest rdf:type tcw:Guest)
                   (?guest -tcw:likes ?wine)]
```

For the first goal, `winesForDinner`, we obtain using `wineForDinner` rule all wines which are possible to be server at the dinner. In our example this refers to `Riesling`, `Merlot`, `Chardonay` and `VinoVerde`. These wines are one of available wines for which exist at least one guest who likes it.

The second goal, `notWinesForDinner`, is to obtain wines which are not liked by at least one guest. This is expressed by the second rule, `notWineForDinner`. For our example the result is: `Retsina`, `Merlot`. We can now for example to know which wines are liked by all guests. This can be obtained by excluding from the list obtained from first goal, common wines obtained by the second goal.

The interest of the committee is to serve at this dinner only wines which are liked by all participants. In order to do this we define a rule expressing which wines are liked by all participants. The rule conclusion is based on the facts which can be derived using the above two rules: `wineForDinner` and `notWineForDinner`.

```
// which available for every ones
[winesToServe: (?wine tcw:winesForAll 'true'^^xs:boolean)
               <-
               (?wine rdf:type tcw:WineForDinner)
               naf(?wine -rdf:type tcw:WineForDinner)]
```

Relating to the above facts, this rule conclude that `Riesling`, `Chardonay` and `VinoVerde` are the appropriate wines to be served at dinner. A simple goal can return these results:

```
[winesToBeServed: <- (?wine tcw:WinesForAll 'true'^^xs:boolean)]
```

6 Related Work

Variables in triples have also been introduced in languages such as N3 [3] and *Jena Rules* [11]. A form of negation-as-failure has been implemented in Jena Rules by using a special built-in predicate. In N3, there is also a form of negation-as-failure, which allows one to test for what a formula does not say, with the help of `log:notIncludes`. But neither N3 nor Jena Rules has a systematic treatment of negative information and open and closed predicates.

7 Testing rules using JenaRulesWeb front-end

The two test cases presented in the 'Cases Study' section are available to be tested online, using an AJAX based application at: `http://oxygen.informatik.tu-cottbus.de/JenaRulesWeb/`.

The `Web Demo for the Jena-Based ERDF Inference Engine of REWERSE I1 v0.1` Application (see Fig. 14) use DWR [1], which allows Javascript in a browser to interact with Java on a server and helps you manipulate web pages with the results.

[1] DWR - http://getahead.org/dwr

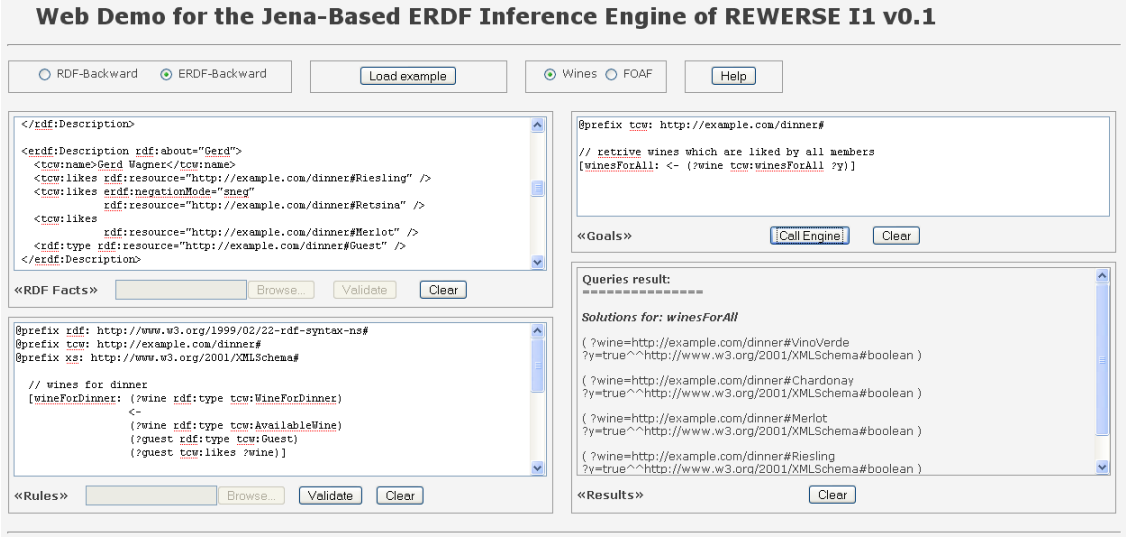

Figure 14: Web Application GUI

The main application is Java based and use ERDF API to inference over a set of facts and a set of rules. It runs under Tomcat 5.5 and Java 1.5 VM. The input data is collected from the text areas of the GUI, then are send to the Java Application, which execute the engine, and finally a HTML based form of the result is displayed in the `Results` panel (see Fig. 17).

The application is available to be used in two modes:

- first mode is RDF(S) based (use RDF facts and Jena rules) and the inference is performed using the Jena API;
- second mode is ERDF based (use RDF+ERDF facts and ERDF triple based syntax of rules) and the inference engine from the ERDF API is used;

We can switch between the two modes by selecting one of the radio boxes, respectively `RDF-Backward` or `ERDF-Backward` (see Fig. 15). For every of the two modes default examples are

Figure 15: Selecting inference engine - RDFS/ERDF

available. In the ERDF mode, the default examples are the ones presented in the 'Cases Study' section of this paper. Every correspondent example for a mode is loaded by pushing the `Load Example` button (see Fig. 15).

The GUI contains three editable areas, where facts, rules and goals are written or modified, and one where the goals results are displayed after the 'Call Engine' button is pushed (see Fig. 16).

Fig. 17 shows partial results obtained for the goals of the FOAF based Case Study.

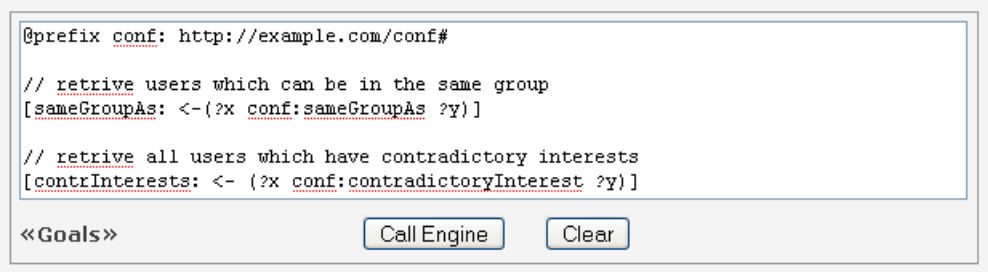

Figure 16: Query panel

Figure 17: Query result for FOAF based Case Study

8 Conclusion and Future work

This document presents an abstract and an RDF-style concrete syntax for ERDF, which allows to represent negative fact statements and supports reasoning with open and closed predicates. We have also argued that these issues are of practical significance by showing how they affect the popular FOAF vocabulary.

Future work will include further extensions of the language, including constructs for handling uncertainty and reliability, and their implementation in the ERDF API.

References

[1] Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damasio, and Gerd Wagner. Negation and Negative Information in the W3C Resourse Description Framework. *Annals of Mathematics, Computing and Teleinformatics*, 1(2):25–34, 2004.

[2] Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damasio, and Gerd Wagner. Stable Model Theory for Extended RDF Ontologies. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *Proceedings of the 4th International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science (LNCS)*, pages 21–36, Galway, Ireland, 6-10 November 2005. Springer-Verlag.

[3] Tim Berners-Lee. N3 (Notation 3). http://www.w3.org/DesignIssues/Notation3.html, 1998.

[4] D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation February 2004. http://www.w3.org/TR/rdf-schema/.

[5] Dan Brickley and Libby Miller. FOAF Vocabulary Specification 0.91. http://xmlns.com/foaf/spec/, November 2007.

[6] Klyne G. and Caroll J.J. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-concepts/.

[7] Frank Van Harmelen Grigoris Antoniou. *A Semantic Web Primer*. MIT Press, 2004.

[8] Object Management Group. Ontology Definition Metamodel. http://www.omg.org/docs/ptc/07-09-09.pdf, November 2007.

[9] Heinrich Herre, Jan O. M. Jaspars, and Gerd Wagner. Partial logics with two kinds of negation as a foundation for knowledge-based reasoning. In D.M. Gabbay and H. Wansing, editors, *What is Negation?* Kluwer Academic Publishers, 1999.

[10] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language. Semantics and Abstract Syntax. http://www.w3.org/TR/owl-semantics/, February 2004.

[11] Dave Reynolds. Jena Rules experiences and implications for rule use cases. In *W3C Workshop on Rule Languages for Interoperability*, 2005.

[12] Gerd Wagner. A database needs two kinds of negation. In B. Talheim and H.D. Gerhardt, editors, *3rd Symposium on Mathematical Fundamentals of Database and KnowledgeBase Systems*, volume 495 of *Lecture Notes in Computer Science (LNCS)*, pages 357–371. Springer-Verlag, 1991.

[13] Gerd Wagner. Web rules need two kinds of negation. In F. Bry, N. Henze, and J. Maluszynski, editors, *Principles and Practice of Semantic Web Reasoning, Proceedings of the 1st International Workshop, PPSWR '03*, volume 2901 of *Lecture Notes in Computer Science (LNCS)*, pages 33–50. Springer-Verlag, 2003.

[14] Gerd Wagner, Adrian Giurca, and Sergey Lukichev. R2ML: A General Approach for Marking up Rules. In F. Bry, F. Fages, M. Marchiori, and H. Ohlbach, editors, *Dagstuhl Seminar Proceedings 05371*, Principles and Practices of Semantic Web Reasoning, 2005.