5 Stanbol Rules

Stanbol Rules is the component aimed to manage and represent rules of inference in the Knowledge Representation and Reasoning layer of the IKS. A rule of inference, or transformation rule, is a syntactic rule or function which takes premises and returns a conclusion. The rule is sound with respect to the semantics of classical logic in the sense that if the premises are interpreted to be true then so is the conclusion. The rule pattern used for representing rules is the *modus ponens*, e.g. ***if*** *condition* ***then*** *consequent*. For example the axiom "every person has a father" can be expressed with the modus ponens in the following way

if X is a person **then** X has a father

and by means of predicate calculus as

$$\forall x.\exists y.Person(x) \Rightarrow hasFather(x,y)$$

where $Person$ and $hasFather$ are two predicates.
The Stanbol Rules component allows to add a layer for expressing business logics by means of axioms, i.e. rules. These axioms can be organized into a container called *Recipe* (Section 2.1), which allows to identify a set of rules that share the same business logic and interpret them as a whole.

As an example Stanbol Rules may help an IKS administrator for defining some integrity check for data fetched from heterogeneous and external sources in order to prevent unwanted formats or inconsistent data. Furthermore Stanbol Rules enable the IKS to derive new knowledge or to perform information integration from different semantically enhanced contents simply defining some rule of inference.

Rules represented with the Stanbol Rules' language can be concretely expressed either as SWRL [HPSB$^+$04] rules or as SPARQL CONSTRUCT [PS08]. The reasons of this double interpretation of the rules is twofold. (i) Adopt different reasoning profiles based on classical DL inference and on RDF querying. When it is required a classical DL reasoning, rules are interpreted as SWRL and for example the Stanbol Rules component can be coupled with the Stanbol Reasoners component. Otherwise, when it is required a transformation of an RDF graph under certain constraints, rules are interpreted as SPARQL CONSTRUCT and are executed by the Stanbol Refactor. (ii) Adopt different rule serializations of Stanbol Rules in SWRL and SPARQL is to ensure the interoperability of the IKS with other rule systems of knowledge bases and also the integration in the IKS of legacy rule systems. In fact SWRL is a proposal of the W3C for a Semantic Web rules-language, combining sublanguages of the OWL Web Ontology Language (OWL DL and Lite) with those of the Rule Markup Language (Unary/Binary Datalog), SPARQL is a recommendation since 2008 for a query language for RDF. The future work in the direction of the interoperability with other rule systems will include the possibility of accepting and exporting also RIF [Kif10] rules.

5.1 Rule Language

In the Stanbol Rules syntax a rule is defined as

$$ruleName[\textbf{body} \rightarrow \textbf{head}] \tag{1}$$

where

- the ruleName uniquely identifies the rule in the rule base and can be any string identifier,
- the body (condition) is a set of atoms or predicates that must be satisfied when evaluating the rule,

- the head or consequent is a set of atoms that must be true if the condition is evaluated to be true.

An atom is the smallest unit of interpretation of a rule, e.g. in predicate calculus the rule $Person(x) \Rightarrow hasFather(x, y)$ has the predicates $Person(\bullet)$ and $hasFather(\bullet, \bullet)$ that are the two atoms in our rule language. Each atom's argument can be a constant, e.g. a URI like <http//dbpedia.org/resource/Bob_Marley>, or a literal such as "Bob_Marley"8sd:string, or a variable that could be any identifier preceded by **?**, e.g. ?x is a variable.
The atoms of Stanbol Rules can be organized in different sets depending on their aim, namely they are:

- Core atoms
- Comparison atoms
- String manipulation atoms
- Arithmetic atoms
- Productive atoms

Core atoms, as we are going to see, allows to design rules that can be expressed both as SWRL and SPARQL CONSTRUCT as they ground on basic predicates based on OWL axioms and directly derives from SWRL-core. Comparison atoms are equivalent to SWRL built-ins for comparison and to SPARQL filters and depending on the task they can be converted either to the former or to the latter. String manipulation and arithmetic atoms are equivalent to SWRL built-ins for strings and maths and to XPath functions implemented by SPARQL. Productive atoms which allow to create new resources into an RDF graph were not provided neither in SWRL nor in SPARQL1.0. The required an extension of ARQ [12], the SPARQL engine provided by Jena [13], in order to enhance the rule system for knowledge production. Productive rules cannot be serialized as SWRL rules, but only as SPARQL CONSTRUCTs, that can be accepted as valid query only by the modified ARQ engine of Stanbol Rules. Below the atoms are analyzed in more detail.

- *Class assertion atom.* It allows to express is-a relations (subsumption) and is identified by the operator $is(classPredicate, argument)$, where
  - $classPredicate$ is a URI that identifies a class
  - $argument$ is the resource that has to be proven to be typed with the $classPredicate$. It can be both a constant (a URI) or a variable.

  The class assertion atom is serialized either into a class atom of SWRL, such as `classPredicate(argument)` or into SPARQL triple pattern of the form `argument rdf:type argument`.

- *Individual assertion atom.* It allows to verify or to assert facts by expressing that an object property holds between two individuals. The syntax for expressing an individual assertion atom is $has(properyPredicate, arg1, arg2)$, where $propertyPredicate$ is the object property that has to be evaluated and $arg1$ and $arg2$ are the two arguments of the property. Each argument of the atom can be either a constant, i.e. a URI, or a variable, e.g. ?x.

[12] http://jena.sourceforge.net/ARQ/
[13] http://openjena.org/

- *Data value assertion atom.* It allows to express and to verify facts where datatype properties hold between individuals and values. A data value assertion atom can be identified by the operator $values(properyPredicate, arg1, arg2)$, where $propertyPredicate$ is the datatype property that has to be evaluated between the $arg1$, which can be either a constant, i.e. URI, or a variable and $arg2$, which can be either a value or a variable.

- *Range assertion atom.* It allows to verify or to assert the range of a certain property. The syntax for expressing range assertion atoms is $range(rangeArg, arg)$, where $propertyPredicate$ is the property whose range is given in arg. Both the arguments of the atom can be either constants or variables.

An extension to core set is composed by the comparison atoms, which basically allow to compare two terms. Namely, they are:

- $same(arg1, arg2)$. It returns true if $arg1$ is equal to $arg2$.

- $different(arg1, arg2)$. It returns true if $arg1$ is different from $arg2$.

- $greaterThan(arg1, arg2)$. It returns true if $arg1 > arg2$.

- $lessThan(arg1, arg2)$. It returns true if $arg1 < arg2$.

- $startsWith(arg1, arg2)$. It returns true if the string associated to $arg1$ starts with the string associated to $arg2$.

- $endsWith(arg1, arg2)$. It returns true if the string associated to $arg1$ ends with the string associated to $arg2$.

The atoms for realizing string manipulation in rules are the following:

- $concat(arg1, arg2)$. It returns a string that is the concatenation of $arg1 + arg2$.

- $substring(arg, start, length)$. It returns the substring of the string arg from the position $start$ for $length$ chars. Both $start$ and $length$ are integers.

- $lowercase(arg)$. It returns the lower case representation of arg, e.g. “Tim Berners-Lee" will be transformed into “tim berners-lee".

- $uppercase(arg)$. It returns the upper case representation of the string in arg, e.g. “Tim Berners-Lee" will be transformed into “TIM BERNERS-LEE".

- $str(arg)$. It returns the literal value of any RDF object,
  e.g. the URI <http://dbpedia.org/resource/Bob_Marley> will be transformed into “http://dbpedia.org/resource/Bob_Marley” and “Bob_Marley”8sd:string will be transforned into “Bob_Marley”.

- $namespace(arg)$. It returns the namespace of any URI as a string,
  e.g. $namespace$(<http://dbpedia.org/resource/Bob_Marley>) will return “http://dbpedia.org/resource/”.

- $localname(arg)$. It returns the local name of any URI as a string,
  e.g. $localname$(<http://dbpedia.org/resource/Bob_Marley>) will return “Bob_Marley”.

The atoms for expressing arithmetic functions are:

- $sum(arg1, arg2)$. It returns a number equal to $arg1 + arg2$.
- $sub(arg1, arg2)$. It returns a number equal to $arg1 - arg2$.

An important point is that the rule system is able to also express production rules in order to create new objects in the knowledge base. For this purpose there are two atoms that allow to bind a value to a variable and to create a new node in the graph with a given identifier (i.e. URI). These atoms are

- $let(arg1, arg2)$, which allows to explicitly assign the values in $arg2$ to the variable in $arg1$. For example the atom $let(?x,$"Tim Berners-Lee"$)$ assigns the value "Tim Berners-Lee" to the variable $?x$. To a variable can be assigned any values either a URI or a literal.
- $newNode(arg1, arg2)$, which allows to create a new RDF dereferenceable via a URI. For example the atom $newNode(?x,$"http://dbpedia.org/resource/Bob_Marley"$)$ creates a new dereferenceable node in the graph, namely <http://dbpedia.org/resource/Bob_Marley>.

The detailed grammar of the language in BNF can be found in Appendix 10.3.

In equation 1 we said that in our syntax a rule is composed by a premise, i.e. body, and a consequent, i.e. head. The body and head are two formulas of conjunctive atoms. The conjunction between two atoms is expressed with a .. The formula in the body always returns a boolean values which states the fact that the atoms in the formula are evaluated to be true in the model, while the formula in the head applies to the model the fact stated by the atoms if and only if the body returns true. As an example we can consider a rule that aims to infer the relation `hasUncle` between the individual `x` and `y` if `z` is the parent of `x` and `z` is the brother of `y`. It will be:

```
uncleRule[has(<http://www.foo.org/myont.owl#hasParent>, ?x, ?z) .
          has(<http://www.foo.org/myont.owl#hasSibling>, ?z, ?y)
             ->
          has(<http://www.foo.org/myont.owl#hasUncle>, ?x, ?y)
]
```

The rule above will be represented in SWRL as it follows

```
<swrl:Variable rdf:ID="x"/>
<swrl:Variable rdf:ID="z"/>
<swrl:Variable rdf:ID="y"/>
<ruleml:Imp>
   <ruleml:body rdf:parseType="Collection">
      <swrl:IndividualPropertyAtom>
          <swrl:propertyPredicate rdf:resource="&eg;hasParent"/>
          <swrl:argument1 rdf:resource="#x" />
          <swrl:argument2 rdf:resource="#z" />
      </swrl:IndividualPropertyAtom>
      <swrl:IndividualPropertyAtom>
          <swrl:propertyPredicate rdf:resource="&eg;hasSibling"/>
          <swrl:argument1 rdf:resource="#z" />
```

```
            <swrl:argument2 rdf:resource="#y" />
         </swrl:IndividualPropertyAtom>
      </ruleml:body>
      <ruleml:head rdf:parseType="Collection">
         <swrl:IndividualPropertyAtom>
            <swrl:propertyPredicate rdf:resource="&eg;hasUncle"/>
            <swrl:argument1 rdf:resource="#x" />
            <swrl:argument2 rdf:resource="#y" />
         </swrl:IndividualPropertyAtom>
      </ruleml:head>
   </ruleml:Imp>
```

The same rule in SPARQL CONSTRUCT will be:

```
PREFIX myont: <http://www.foo.org/myont.owl#>

CONSTRUCT { ?x myont:hasUncle} ?y }
WHERE { ?x myont:hasParent ?z .
        ?z myont:hasSibling ?y}
```

Details about SWRL and SPARQL are beyond the aim of this document and we refer interested readers to [HPSB+04] for the former and to [PS08] for the latter .

5.2 Rules and Recipes

The main added value of Stanbol Rules is the way in which it organizes and represents rules. In fact, one of the key design concept in Stanbol Rules is the *Recipe*, basically a set of rules with an associated URI, which allows to organize rules by the business logic they address.
In other words, a recipe is a reusable set of rules that can be uniquely identified, stored, and accessed within the IKS stack. A single rule is also uniquely identifiable in KReS, but it can be shared by several recipes. The idea is that the recipe can be composed by using rules as ingredients and applied as a single inference object for specific purposes.
As an example we may want to organize rules in a recipe in order to make some inference about kinships. For this purpose we want to use the *uncleRule* of the previous example always together with another rule, *brotherRule*, which aims at inferring the relation `hasBrother` between the individual `x` and `y` if `x` and `y` are both child of `z`, namely

```
brotherRule[has(<http://www.foo.org/myont.owl#hasFather>, ?x, ?z) .
            has(<http://www.foo.org/myont.owl#hasFather>, ?y, ?z) .
               ->
            has(<http://www.foo.org/myont.owl#hasBrother>, ?x, ?y)
]
```

In order to apply those rules to our knowledge base, the rules need to be composed into a recipe, e.g. we can create the recipe `http://kres.iks-project.eu/ontology/meta/rmi_config.owl#MyRecipe` which is composed by `uncleRule` and `brotherRule`. The management of the recipes is performed through an OWL ontology, which allows to represent rules and recipes and relate the former to the latter. Our example can be encoded as: