PEARL: ProjEction of Annotations Rule Language, a Language for Projecting (UIMA) Annotations over RDF Knowledge Bases

Maria Teresa Pazienza, Armando Stellato, Andrea Turbati

ART Research Group, University of Rome, Tor Vergata, Italy
{pazienza, stellato, turbati}@info.uniroma2.it

abstract

In this paper we present a language, PEARL, for projecting annotations based on the Unstructured Information Management Architecture (UIMA) over RDF triples. The language offer is twofold: first, a query mechanism, built upon (and extending) the basic FeaturePath notation of UIMA, allows for efficient access to the standard annotation format of UIMA based on feature structures. PEARL then provides a syntax for projecting the retrieved information onto an RDF Dataset, by using a combination of a SPARQL-like notation for matching pre-existing elements of the dataset and of meta-graph patterns, for storing new information into it. In this paper we present the basics of this language and how a PEARL document is structured, discuss a simple use-case and introduce a wider project about automatic acquisition of knowledge, in which PEARL plays a pivotal role.

Keywords: UIMA, Ontology Development, Knowledge Acquisition

1. Introduction

Nowadays, it is possible to easily access huge volumes of information: however, this information does not appear as an homogeneous stream of data, and instead comes from very different sources and follows heterogeneous patterns that are very difficult to manipulate, use and organize without the help of dedicated tools. Modern Web paradigms, such as the Semantic Web (Berners-Lee, Hendler, & Lassila, 2001), and its associated initiatives such as Linked Open Data (Bizer, Heath, & Berners-Lee, 2009) may support the organization, filtering and search of information thanks to data modeling principles and query facilities, however, much of the Web is still (and will always be) composed of traditional unstructured content, such as text, video, audio and multimedia material in general.

To be able to cope with this huge volume of information, Information Extraction (IE) engines allow for lifting of relevant data from heterogeneous information sources and for its projection towards predefined knowledge schemes, thus enabling advanced access based on semantic rather than textual indexing.

The success of semantic search engines such as Eqentia[1] or Evri[2] and Information Extraction services such as OpenCalais[3] and Zemanta[4] show that there is large demand for these solutions, while software platforms such as UIMA[5] (Ferrucci & Lally, 2004), GATE[6] (Cunningham, 2002) or Ellogon[7] (Petasis, Karkaletsis, Paliouras, Androutsopoulos, & Spyropoulos, 2002) provide the middleware for designing and implementing the extraction process under a clearly defined practice.

However, the above engines are oriented towards provisioning of APIs and services for producing knowledge modeled according to open standards, but they fail in allowing users to create the definition of new content extractors and annotators, and sometimes the users are not even able to access the code of the engines implicitly available through the provided services. On the other side, the cited middleware platforms for text engineering support development of IE systems but do not provide guidance/facilities on how to store this information. It clearly appears that though research on Natural Language Processing and Information Extraction have found an industrial standardization, we lack of clearly defined specifications and tools on how to use this information to create data.

To meet this need, we have addressed two of the most popular standards for information extraction and knowledge representation publication now available: respectively, UIMA and RDF, and defined a language, based on these standards, for supporting and facilitating the acquisition of knowledge starting from raw unstructured information to obtain widely accessible datasets.

The above language, PEARL, the ProjEction of Annotations Rule Language, allows for the transformation and projection of information modeled according to the Unstructured Information Management Architecture (UIMA), onto RDF Datasets.

UIMA data comes in the form of annotations taken over unstructured content (of any format and nature) and modeled as feature structures (Carpenter, 1992). In PEARL, UIMA annotations are analyzed through pattern-matching rules, their elements (feature structures) are then processed, transformed, matched against the target semantic repositories, and then finally transformed into RDF triples according to the vocabulary of the adopted ontology and modeling language.

The paper is organized as follows: section 2 introduces the syntax of the language and the structure of a PEARL projection document. In section 3 the main features of the

[1] http://www.eqentia.com/
[2] http://www.evri.com
[3] http://www.opencalais.com/
[4] http://www.zemanta.com/
[5] http://uima.apache.org/
[6] http://gate.ac.uk/
[7] http://www.ellogon.org/

languages are presented and detailed. Section 4 discusses an application case of PEARL. The following section introduces the CODA architecture: a wider project about knowledge acquisition of which PEARL is a central aspect. Section 6 concludes the paper.

2. PEARL Document Organization

PEARL, ProjEction of Annotations Rule Language, is an easy-to-learn, yet powerful, language for projecting UIMA annotations over RDF triples. A simplified version of the grammar behind it, expressed in Backus-Naur form, can be seen in Fig. 1.

Each projection document (a document containing PEARL rules) can be considered as divided into two main and distinct parts: in the upper part there is the listing of all the namespaces-prefixes that will be used in the projection rules concerning ontology resources, and the second part, which is normally longer, contains the projections rule. Here follows the description of these sections.

```
prRules := prefixDeclaration* prRule+ ;
prefixDeclaration := prefix '=' namespace ';';
prRule := 'rule' uimaTypePR (ID ':' idVal)? Conf?
        ('dependsOn' (depend)+)? '{' alias? nodes?
        graph where? parameters? ';'? '}';
depends := DependType '(' idVal ')';
alias := 'alias' '=' '{' singleAlias+ '}' ;
singleAlias := idAlias uimaTypeAndFeats ;
nodes := 'nodes' '=' '{' node+ '}' ;
node := idNode type (uimaTypeAndFeats | condIf);
condIf := 'if' condValueAndUIMAType condElseIf*condElse?;
condValueAndUIMAType := '(' condBool ')' '{'
        uimaTypeAndFeats | OntoRes'}';
condElseIf := 'else if' condValueAndUIMAType ;
condElse := 'else' '{'uimaTypeAndFeats | OntoRes'}';
graph := 'graph' '=' '{' triple+ '}' ;
triple := tripleSubj triplePred tripleObj
        | 'OPTIONAL' '{' tripleSubj triplePred tripleObj '}';
where := tripleSubj triplePred tripleObj
        | 'OPTIONAL' '{' tripleSubj triplePred tripleObj '}';
parameters := 'parameters' '=' '{' (parameterNameValue
        (',' parameterNameValue)*)?'}';
parameterNameValue := parameterName ('='parameterValue)? ;
```

Fig. 1 BNF of (part of) the grammar used in the Projection Rule File

2.1. Prefix Declaration

The first part of a projection document contains all the ontology prefixes being used in the projection rules.

Note that these prefixes may not be the same (though they may overlap) of those which have been declared inside the target ontology and are independent from that declaration. They thus are local to the projection process and they are used to expand prefixed names inside the document into valid RDF URIs and no trace of them is left in the target ontology. The use of these prefixes is highly encouraged to have a more readable document.

2.2. Projection Rules

After the prefix declaration, the rest of a projection document contains the list and description of each projection rule. Each Projection Rule is divided into the following parts (some of them are optional): a rule declaration, followed by its definition, which is in turn composed of the following sections:

- nodes
- graph
- where
- parameters.

A brief explanation is provided of every one of them.

2.2.1. Rule declaration

A rule starts with a declaration, expressed through the keyword *rule* and is concluded with a curly bracket "{", initiating its definition.

Each rule is associated to a UIMA type from the adopted UIMA Type System, thus the type is the first element in the declaration: any UIMA annotation taken after that type (written following the UIMA standards regarding types, i.e. a java-like dot-separated package name followed by a Capital word referring to the associated UIMA Type) will trigger the possible use of this rule.

After the type declaration, there is an optional rule identifier that can be used to make references to a resource (placeholder or variable) from other rules, according to different relationship of dependency (explained later on this paper).

A number in the range of 0..1 follows, representing a confidence value which can be used to rank different rules associated to the same UIMA type. This value can be important to the application parsing these rules, as it may give a first ranking for the rule to use given a specific annotation type.

The declaration may end with a list of dependencies to other rules. Each dependency must specify the type and the kind of relationship which is established.

2.2.2. Alias

An alias is a compact way to use a value which is presented inside a feature of the given annotation type. An alias is denoted by a '$' followed by its name.

2.2.3. Nodes

The third part, which is optional only if the rule depends on another rule (see section 3.3), provides a list of placeholders for ontology nodes from UIMA annotations.

These placeholders are used to state which features of the triggered UIMA type are important for the target ontology, and to specify which kind of RDF nodes (URI, typed literal, plain literal) will be used as recipients to host the information that will be projected from them.

A set of operators are available for applying different transformations to the features, projecting them onto valid RDF nodes. Default conversions are applied when no operator is specified, and they are inferred on the basis of the target RDF node type (e.g. if the node type is an URI, the feature value is first "sanitized", to remove characters which are incompatible with the URI standard, and then used as a local name and concatenated to the namespace of the target ontology to create an URI). Specific transformations can be invoked for producing URI according to different formats (e.g. reified emails) which

do not need to conform to the baseuri of the adopted ontology.
In UIMA it is possible that the value of a particular feature is a type itself, thus containing other features, and so on recursively, as stated in the feature structure theory (Carpenter, 1992): `FeaturePath` is a standard notation introduced in UIMA to identify arbitrary values in a complex path describing a specific traversal of a feature structure. The standard feature path presents some limitations, which PEARL tries to overcome (see paragraph 3.2).
Sometimes it may be necessary to determine a different assignment of one feature (e.g. projecting it onto an OWL Class or onto a Property) depending on the value of another feature. This can be specified by using the alias mechanism and a simple *if/else* construct (see the grammar, the rule for *condBool* in Fig. 1, and the example in paragraph 3.4). So for example it is possible to assign the OWL class *Male* or *Female* to the placeholder *gender* by checking a value of a feature (e.g. if that feature has the value 'mr.' *Male* is assigned; conversely *Female* is assigned for the value 'mrs.').

2.2.4. Graph

The graph section contains the true projections over the target ontology graph, by describing a graph pattern which is dynamically populated with grounded placeholders and variables (defined in the WHERE part).
The graph pattern consists of a set of triples, where the first element is the subject, the second is the predicate and the third the object of an RDF statement. Each single element in the graph may be one of the following: a placeholder, a variable, an RDF node or an abbreviation.
Inside a graph pattern, placeholders (defined in the nodes section of the current rule or of other referenced projection rules), are identified by the prefixed symbol "$". RDF nodes can be references in graph patterns through the usual notation for URIs ("< and >" delimit standard URIs) or by prefixed local names as normal.
The abbreviations are represented by a finite list of words that can be used in place of an explicit reference to RDF resources. An example of such abbreviation is the character *a* interpreted as *rdf:type*
Finally, it is possible to use variables (by prefixing their names with a "?" symbol) when there is need to dynamically reference an RDF node already existing in the target ontology, which is not known in advance (i.e. it is not statically added in the rule, but dynamically retrieved from the ontology by means of unification, see next section for more details).

2.2.5. Where

As for the graph section, the where section contains a graph pattern: this pattern is matched over the target ontology to retrieve nodes already existing in the target ontology by means of variable unification, so that the variables substitutions can be reused in the graph section.
The purpose of this graph is to be able to link newly extracted data with information which is already present in the target ontology. In this sense, it is close to the purpose of the where statement in a SPARQL CONSTRUCT query. The unification mechanism allows to assign values to variables by constraining them on the basis of information which is thought to be present in the ontology: these substitutions are then applied to the graph pattern of the graph section to project the data in the over target ontology.

2.2.6. Parameters

The fifth and last part, optional, consists of a list of parameters. A parameter can be in the form of a "<name,value>" pair or just a name.
There is no pre-assigned semantics to any parameter, they are just outputted by any rule when it is being applied, and their meaning is properly interpreted by specific application components (such as CODA component implementations, see paragraph 5) which maybe associated to a given projection document.
These parameters can thus be seen as flexible extension points for the language, requiring no dedicated syntax, and conveying specific information (parameter values can contain placeholder/variable assignments) for the appropriate listener.

3. Features of the Language

As the main structure of a PEARL document has been described, we can now present the different features of this language. In this section we describe them and we show how they can be used. The features which are being presented are:

- how to deal with missing values in the current annotation
- how to retrieve all the values from a list in a UIMA feature
- establishing a dependency among two rules
- how to assign a value to a placeholder depending on another value in a different feature
- how to retrieve RDF resources from the target ontology

3.1. Missing values in the annotation

An annotation in UIMA is a complex structure, normally containing more than one feature and each feature can be a complex structure itself (i.e. a feature structure). It may happen that not all features have a value associated to them. In this case the placeholder which receives the missing value is not ground.
When this placeholder is being used to compose an RDF triple, the application of such a rule should thus *fail* because it is impossible to instantiate the RDF graph pattern.

```
my=http://art.uniroma2.it/imdb#;
www=http://www.movieontology.org/2009/11/09/;
dbp=http://dbpedia.org/ontology/;
movie=http://www.movieontology.org/2010/01/movieontology.owl#;

rule it.uniroma2.art.imdb id:film {
   nodes ={
      filmId     uri                    _it.uniroma2.art.imdb:title
      filmTitle  literal(xsd:string)    _it.uniroma2.art.imdb:title
      year       literal(xsd:integer)   _it.uniroma2.art.imdb:year
      descr      literal(xsd:string)    _it.uniroma2.art.imdb:description
   }
   graph ={
      $filmId     a                     www:Movie .
      $filmId     movie:title           $filmTitle .
      $filmId     movie:releasedate     $year .
      OPTIONAL{ $filmId   my:description   $descr} .
   }
}
```

Fig. 2 Use of the OPTIONAL tag

Sometimes the occurrence of a particular RDF triple is not mandatory and if it there is any issue (e.g. a null value) in grounding it, the other triples in the graph could still be used to populate the target Dataset. In this case this non-mandatory triple is surrounded in the projection rule by the word OPTIONAL. This signals that the other RDF triples are independent from this one and can still be suggested even if this particular one has not been instantiated.

The OPTIONAL modifier in the graph may recall the OPTIONAL in SPARQL SELECT queries; however here the semantics differ in that this refers to the success of the projection operation: the graph does not need to be matched against the target ontology, but instead to be written into it; in this case, satisfying the graph is considered as satisfying the set of all write operations on each triple. A write operation succeeds if all the three elements of its triple are bound (instantiated). The OPTIONAL modifier here is similar to the one in the WHERE: the whole writing of the graph pattern is not compromised if the triples inside an OPTIONAL clause fail to be written (they are not completely instantiated), and these are simply left out from the global write of the graph.

In the example in Fig. 2 the RDF triple adding a description to a movie is tagged as OPTIONAL, because not all movies have an associated description.

Another situation in which it can be useful to tag a triple with the OPTIONAL is when one of the placeholder is not empty just because the relative feature is empty itself, but when a feature path in some situation (an instance of an annotation) leads to a "dead end" (this annotation do not have a complex value associated to a feature presented in the path, instead it is null).

3.2. Accessing all the values of a list

Each value inside an annotation is identified by an univocal path, called feature path. This path represents all the features which must be navigated to access the desired value. If one of the feature in the path is a list, the standard used in the feature path states that the exact element one is interested in, must be used. PEARL adds

```
www=http://www.movieontology.org/2009/11/09/;
ontology=http://dbpedia.org/ontology/;
   nodes ={
      filmId    uri       _it.uniroma2.art.imdb:movieId
      actId     uri       _ it.uniroma2.art.imdb:actorsList/personId
   }
   graph = {
      $filmId     a                    www:Movie .
      $actorId    movie:title          ontology:Actor .
      $actId      movie:isActorIn      $filmId .
   }
}
```

Fig. 3 Accessing all the values in a List

the possibility to extract all the values in a list and associate them to a single placeholder (in this case the placeholder will have naturally more than one value).

The syntax for doing so is the same as the one used to navigate a feature with a single value (simple or complex one), it uses the name of the feature, which contains the list and do not specify any particular index. This can be used not just for the last feature in a feature path, but even for one in the middle of the path. In this case the path will generate "subpaths" which are navigated one by one and all the value are stored in a single placeholder.

In the example presented in Fig. 3 the second placeholder, *actId*, does not contain just a single value, but a list of values. As the name of the feature may suggest, *actorList* contains a list of values (and these values are not primitive values, such as string or integer). It is also possible to use a specific position inside the List/Array by using the standard syntax *actorList[i]*, where *i* stands for the position where the value is stored (starting from 0 and not from 1).

3.3. Establishing a dependency among two rules

Each projection rule is associated to a specific UIMA type, so it can access all the information present in the annotation which triggered its use. This in certain case can be seen as a limitation as the elements of different types

```
www=http://www.movieontology.org/2009/11/09/;
movie=http://www.movieontology.org/2010/01/movieontology.owl#;
rule it.uniroma2.art.imdb id:film {
   nodes = {
      filmId    uri    _it.uniroma2.art.imdb:movieId
   }
   graph = {
      $filmId          a          www:Movie .
   }
}

rule it.uniroma2.art.imdbCast dependsOn last(film){
   nodes = {
      actorId uri  it.uniroma2.art.imdb:actorsList/personId
   }
   graph = {
      $actorId         a                    ontology:Actor .
      $actorId         movie:isActorIn      $film:filmId .
   }
}
```

Fig. 4 Dependency among rules

```
my=http://art.uniroma2.it/imdb#;
xsd=http://www.w3.org/2001/XMLSchema#;
owl=http://www.w3.org/2002/07/owl#;

rule it.uniroma2.art.uima.imdb.Animal {
    alias = {
        animalType    _it.uniroma2.art.uima.animal:type
    }
    nodes = {
        animalId      uri     _it.uniroma2.art.animal:animalId
        animalClass uri     if(animalType == reptile){
                                my:Reptile
                            }else if(animalType == insect){
                                my:Insect
                            }elsef
                                my:Animal
                            }
    }
    graph =f
        $animalId      a      $animalClass .
    }
}
```

Fig. 5 Use of if/else in a rule

may need to be composed together.

To overcome this limitation it is possible to state a dependency between two or more projection rules. The example in Fig. 4 shows how.

By first, the rule which the other one depends on must have an id (the first one has *film* as id). The other one states its dependency using the keyword *dependsOn* followed by the type of dependency and the id of this rule. There are two main families of dependency, introduced by the keywords *dependsOn* and *imports.* The difference is that the former states that the rule this one is depending on should have produced some suggestions (no problems were found in the no OPTIONAL triples), while the latter does not require this check.

In this example *last* is used as the dependency type. This means that when the CODA will use the second rule it will look back to the other annotation until it finds when and where the first rule was used for the last time.

At this point CODA will consider this other annotation as the target of this particular instance of dependency, so the application of the second rule for the given annotation depends on the other annotation just found.

Once the "link" between these two rules has been established the rules that stated the dependency is now able to use the placeholder defined and initialized in the other rules.

The syntax to use the other placeholder is quite similar to using a local placeholder, the only different is that before the placeholder name, but after the "$" symbol, one must use the other rule's id followed by ":".

The second rule use the placeholder *filmId* from the first rule by writing *$film:filmId* in its second suggested triple.

PEARL support different type of dependency, such as *lastOneOf*, and we are implements others as well.

3.4. Dynamic assignment to a placeholder

Sometimes a value is assigned to a placeholder depending on another information (value) contained in a different feature. This can be accomplished in PEARL using the *if/else* construct and the alias section.

```
www=http://www.movieontology.org/2009/11/09/;
movie=http://www.movieontology.org/2010/01/movieontology.owl#;
xsd=http://www.w3.org/2001/XMLSchema#;
owl=http://www.w3.org/2002/07/owl#;

rule it.uniroma2.art.imdb id:film{
    nodes = {
        filmId      uri                 _it.uniroma2.art.imdb:movieId
        filmTitle   literal(xsd:string)   _it.uniroma2.art.imdb:title
    }
    graph = {
        $filmId          a              www:Movie .
        filmId           movie:title    $filmTitle .
    }
    where = {
        ?filmId          a              $filmTitle .
    }
}
```

Fig. 6 Use of SPARQL in a rule

In this example the value contained in the placeholder *animalClass* depends on the value contained in the feature *it.uniroma2.art.uima.Animal:type*.

The placeholder *animalClass* is going to contain a different resource after the *if/else* is evaluated for every single annotation that triggers the use of this rule. In this case an already existing resource (class) is used but a UIMA feature could have been easily used.

The *if/else* mechanism bring to CODA the possibility to have a dynamic assignment to each placeholder.

3.5. Interacting with the ontology

PEARL gives the possibility to interact with the ontology to retrieve already existing RDF resource. This interaction is done using SPARQL (only a subset of SPARQL syntax have been implemented).

In the where section of this example a variable, *filmId*, is defined (it is a variable and not a placeholder because it starts with"?" and not with "$").

The simple SPARQL query tries to find out the id of the movie with a particular title. If it not able to find it then CODA will use, if present, a placeholder with the same name as the variable.

So if there is not a movie with that particular title in the ontology then a new movie is created with a given title and description (if present). Otherwise the already existing one is used and the description is added.

In may seem strange that the first two triples are suggested because the movie appears to already exist as that information has just been used in the SPARQL query. This is not an error because the triple store underneath the external program should not have any problem in adding a triple which is already present (or even the external program should decide not o add the triple).

4. A real case Projection Rules files

In Fig. 7 we show an example of a typical use (the projection rules file has been reduced to be used in this paper) of the PEARL language to project information extracted from the Internet Movie Database (IMDB) site[8].

[8] http://www.imdb.com/

A UIMA annotator has extracted information about movies, tv series and actors performing inside them. In this document, 4 PEARL rules have been defined to project this information over RDF triples.

In the nodes section of each of the rule we can see how the different elements from the extracted annotations are processed and projected as RDF nodes of different nature (URI or literal, plain or typed), which are used to fill homonymous placeholders in the subsequent graph sections. Note that even a simple declaration of the nature of the node (e.g. being it an URI) implies some kind of transformation of the element. For instance, when the type is an URI, the feature value is – by default – "sanitized" and used as the local name of a URI composed over the chosen namespace for the target RDF. Further transformations are possible, though operators provided by the CODA framework (see paragraph 5); for instance, if uri(mail) is specified in the nature of the node, the transformer expects a compatible mail address to be found in the value, and it further normalizes it as a *mailto:* URI.

Custom transformers can also be added to the framework though OSGi[9] extension points, while the language hosts them and recognize them though use of qualified names (e.g. namespace + id, or prefixed notation).

As we can see in Fig. 7, in the first rule the value contained in the feature *it.uniorma2.IMDBFilm:movieId* is transformed into an URI and assigned to the placeholder *movieId*.

Each element of an RDF triple defined in the *graph* section of a rule can be an explicitly referenced RDF node (URI, literal or blank node), a placeholder or a variable the value of which is obtained through a SPARQL query defined in the same rule using PEARL itself: in the example, the *movieId* placeholder is used as subject of four different triples in the first rule.

In the first rule, five RDF triples are suggested, the first four are "mandatory", this means that if at least one of them cannot be instantiated with proper values, then this rule for the particular annotation will not produce any suggestion. The fifth triple is tagged with OPTIONAL, and, as said before, if it has any problem, then the other triples can still be used.

To navigate inside the features of a UIMA annotation, we used the extended version of the UIMA `FeaturePath`[10] language. For instance, in the third rule we assign to the placeholder *starSite* the value of the feature *site* contained inside the feature *it.uniorma2.imdb.IMDBStar:imdbSite*.

These four rules can be divided into two categories:

- the first two are independent from the others
- the last two have two different type of dependency: *lastOneOf* and *last*)

Using the dependency type *last*, the fourth rule is able to use a placeholder defined in the last application of the second rule (the one identified by the id *series*). It uses the

```
my=http://art.uniroma2.it/imdb#;
xsd=http://www.w3.org/2001/XMLSchema#;
www=http://www.movieontology.org/2009/11/09/;
rule it.uniroma2.IMDBFilm id:film {
   nodes = {
      movieId   uri          _it.uniroma2.IMDBFilm:movieId
      filmTitle  literal(xsd:string) _it.uniroma2.IMDBFilm:title
      year       literal(xsd:integer) _it.uniroma2.IMDBFilm:year
      rate        literal(xsd:float) _it.uniroma2.IMDBFilm:score
      desc       literal(xsd:string) _it.uniroma2.IMDBFilm:descr
   }
   graph = {
      $movieId          a                        www:Movie .
      $movieId          movie:title              $filmTitle   .
      $movieId          movie:releasedate        $year
      $movieId          my:imdbsite              $rate.
      OPTIONAL { $movieId  my:movieDescription $desc } .
   }

rule it.uniroma2.IMDBTVSeries id:series{
   alias = {
      year  _it.uniroma2.art.uima.imdb.IMDBFilm:endYear
   }
   nodes = {
      movieId      uri  _it.uniroma2.IMDBFilm:movieId
      seriesName  literal(xsd:string)  _it.uniroma2.IMDBFilm:title
      startYear    literal(xsd:integer) _it.uniroma2.IMDBFilm:year
      endYear      literal(xsd:integer) if($year ==0){
                          _it.uniroma2.IMDBFilm:notExisting
                                  } else{
                          _it.uniroma2.art.IMDBFilm:endYear}
      rate          literal(xsd:float) _it.uniroma2.IMDBFilm:score
      desc          literal(xsd:string)  _it.uniroma2.IMDBFilm:desc
   }
   graph = {
      $movieId          a                        my:TVSeries .
      $movieId          movie:title              $seriesName  .
      $movieId          movie:releasedate        $startYear  .
      OPTIONAL { $movieId        my:endedIn $endYear }
      $movieId          movie:imdbrating         $rate .
      $movieId          my:movieDescription      $description .
   }
}

rule it.uniroma2.IMDBStar id:star dependsOn
            lastOneOf(movie, film, series) {
   nodes = {
      starId      uri                  _it.uniroma2.IMDBStar:personId
      starName  literal(xsd:string)  _it.uniroma2.IMDBStar:name
      starSite    literal(xsd:string)
                              _it.uniroma2.IMDBStar:imdbSite/site
   }
   graph = {
      $starId             a                        my:Star     .
      $starId             my:imdbsite              $starSite    .
      $starId             my:personName            $starName  .
      $movie:movieId   my:hasStar $starId      .
   }
}

rule it.uniroma2.IMDBCreator id:creator dependsOn
            last(series){
   nodes = {
      creatorId       uri      _it.uniroma2.IMDBCreator:personId
      creatorName  literal(xsd:string)
                          _it.uniroma2.IMDBCreator:name
      creatorSite     literal(xsd:string)
                          _it.uniroma2.art.IMDBCreator:imdbSite/site
   }
   graph = {
      $creatorId          a                        my:Creator .
      $series:movieId     my:createdBy             $creatorId .
      $creatorId          my:imdbsite              $creatorSite          .
      $creatorId          my:personName            $creatorName          .
      $creatorId          my:isCreatorOf           $series:movieId .
   }
}
```

Fig. 7 Example of a Projection Rule File

[9] http://www.osgi.org/Main/HomePage

[10] Standard FeaturePath does not support generic reference to whole collections such as arrays, lists etc… but only to their specific values (the domain of FeaturePath consists of the set of final values in the addressed Feature Structure).

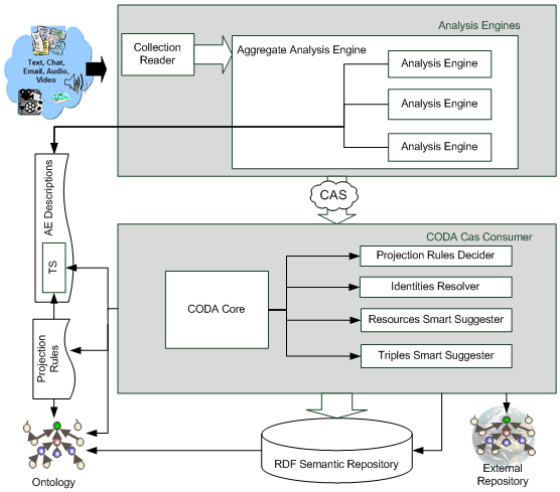

Fig. 8 CODA Architecture

placeholder *movied* defined and instantiated in the other rule (in its last application) by writing *$series:movieId.*

Using the dependency *lastOneOf*, the third rules is stating that it is interested in the last application of the first or the second rule (the one that is nearest to its own usage). Inside its own *graph* section it then uses the placeholder *movieId*, which both the dependency rules have, by using the construct *$movie:movieId* (similar to what the forth rule is doing).

The order among annotation is very important when there is a dependency (for example *last* and *lastOneOf*) and PEARL uses the order provided by UIMA regarding the begin and end of each annotation.

Once we have presented the syntax of PEARL and the associated grammar, its features, and described the characteristic of each rules in the example, we can now describe what each rule does, from a general point of view, to understand how to write a projection rules file given the annotation types provided by a UIMA Analysis Engine (for a complete description of the UIMA architecture please refer to the official site[11]).

The first rule is invoked when an annotation regarding a movie is found. Each annotation has its own type, so it is easy to spot what each annotation contains. Each movie in the imdb site has an id, a title, a year when it was released, a score (given by users) and a description. Not all movies are provided with a description, so it is important to remember to tag every RDF triples in the *graph* section of the rule that deals with the placeholder containing the movie's description. Then in its own graph section the relative RDF triples are suggested (the RDF resource which represents a movie identified by a particular id is an instance of the class Movie, was released in a specific year and it may have a description). The second rule deals with TvSeries. The UIMA annotator used for this example have a peculiarity: if the series does not have an end year (it is still shown on the TV) then it give the value 0 to the feature of the annotation which should store the end year. To manage this aspect in PEARL we used the *if/else* mechanism (and the alias section). If the value contained in the alias *year* is equal to 0, then we put in the placeholder *endYear* a value from a feature that does not exist and by using the OPTIONAL tag, we are able not to add the wrong year to the ontology as the end year of this Tv series. If we do not use this approach, then all the series which are still airing on the TV would have zero as the end year and this can generate some errors in the ontology.

The fourth rule focus its attention on the annotations regarding the creators of television series. These people are defined only if they have created at least a series. The UIMA annotator do not provide any information in the annotation itself about which series the person has created, so this rule needs to depend on the last TV series annotated by the annotators (and the *last* dependency is used). This approach can be used because the imdb pages have a specific structure, in which first the title of the television series or movies is presented and then the list of creators, directors, actors, stars follows. Inside its own *graph* section the fourth rule uses the placeholder defined in the other rules.

The third rule is invoked when the stars (main actors) of a movie or a tv series are found (annotated). The peculiarity of this kind of annotation is that both movies and series have stars, so this rules should behave a little different according to which type of show the star played in. This is achieved with the dependency *lastOneOf*, which provides an easy way to use a placeholder taken from two different rule and use it without the need to replicate the relative suggested RDF triples (both rules must have a placeholder with that name).

5. CODA

PEARL was developed inside the CODA architecture (Computer-aided Ontology Development Architecture).

The motivation behind CODA lies in the gap between the large availability of Information Analysis components for different frameworks (such as UIMA[12] (Ferrucci & Lally, 2004) and GATE[13] (Cunningham, 2002)), and the non-immediateness in exploiting their output – normally structured annotations – to be fed to a knowledge base.

The CODA Architecture, Fig. 8, previously presented in (Fiorelli, Pazienza, Petruzza, Stellato, & Turbati, 2010)foresees a series of components addressing all typical issues related to knowledge acquisition and providing all required facilities needed for this task: access both to the target knowledge base and to external semantic repositories, consuming Entity Naming Services (for example (Bouquet, Stoermer, & Bazzanella, 2008)), resolving identity among generated entities, access to structured annotations from IE systems, separation of finer refinement steps for knowledge acquisition (triple generation, re-classification of entities, human validation, feedback etc…) are all facilities provided by CODA components. CODA architecture sits on top of the UIMA

[11] http://uima.apache.org/

[12] http://uima.apache.org/

[13] http://gate.ac.uk/

standard for Unstructured Information Management (on top on Fig. 8) and extends this architecture by exploiting its output for creating new knowledge to be fed to semantic repositories. From the UIMA point of view CODA can be considered as a CAS Consumer, because it takes as input the CAS produced by one or more Analysis Engines (AE), uses the information contained in the annotations to develop or enrich an ontology.

CODA is also a concrete framework[14] (modeled after its homonymous architecture) providing a software infrastructure for coordinating CODA components and implementations of some of them.

One of the core elements of CODA is exactly PEARL, which is used for projecting UIMA Annotations over RDF triples.

To parse a Projection Rules file by CODA, we used ANTLR[15] (Parr & Quong, 1994), a tool which given in input a grammar written in EBNF (Extended Backus-Naur Form), an extension to BNF (Backus-Naur Form), generates the relative Java classes to parse a file written using the defined grammar. CODA uses these classes to parse the Projection Rules file to construct an internal model to speed up the process of using the rules.

CODA as been used in several different domain (from the movie domain to the agricultural one) with good results.

Because CODA in completely integrated in the UIMA architecture, it allows, and it encourages, the reuse of already existing and tested AEs whenever is possible. This approach reduced the time in developing an application to populate an ontology, using AEs which are able to deal with the desired domain. The user/application needs, in most of the cases, just to write the projection rules which suggest to CODA how to navigate inside the UIMA annotations, which information to extract and how to add the new resources to the ontology.

In some cases it may be necessary to modify the standard behavior of CODA on how to create a new resource, the discovery of already existing ones, or how and which RDF triples should be suggested. This personalization of CODA can be achieved by implementing specific version of the modules presented in the architecture: the Projection Rule Decider, the Identity Resolver, the Resources Smart Suggester and the Smart Triples Suggester.

In this paper we do not provide a description and the goals of each component, because we focus our attention at PEARL, we just want to present the possibility of personalizing CODA for the target domain (ontology and AEs). This personalization is achieved by using Felix Apache[16], an implementation of the OSGi[17] specifications (release 4), which provides a mechanism to add at runtime Java code (in CODA every component uses this mechanism, so every component can be replaced with a new version, without the need to change any line of code of CODA).

[14] http://art.uniroma2.it/coda
[15] http://www.antlr.org/
[16] http://felix.apache.org/site/index.html
[17] http://www.osgi.org/About/HomePage

6. Conclusion

In conclusion this paper presents a language which is able to navigate into complex UIMA annotations, to extract the desired information and then to construct RDF triples using what users have written in the projection rules file. It is a simple to use language and, even if it was developed inside the CODA architecture and framework, it can be used in other applications. We are adding more features to PEARL to extend its own capabilities and possible uses.

7. References

Berners-Lee, T., Hendler, J. A., & Lassila, O. (2001). The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American , 279* (5), 34-43.

Bizer, C., Heath, T., & Berners-Lee, T. (2009). Linked Data - The Story So Far. (T. Heath, M. Hepp, & C. Bizer, Eds.) *International Journal on Semantic Web and Information Systems (IJSWIS), Special Issue on Linked Data , 5* (3), 1-22.

Bouquet, P., Stoermer, H., & Bazzanella, B. (2008). An Entity Naming System for the Semantic Web. *In Proceedings of the 5th European Semantic Web Conference (ESWC 2008).* Springer Verlag.

Carpenter, B. (1992). *The Logic of Typed Feature Structures. Cambridge Tracts in Theoretical Computer Science* ((hardback) ed., Vol. 32). Cambridge University Press.

Cunningham, H. (2002). GATE, a General Architecture for Text Engineering. *Computers and the Humanities , 36*, 223-254.

Ferrucci, D., & Lally, A. (2004). Uima: an architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng. , 10* (3-4), 327-348.

Fiorelli, M., Pazienza, M. T., Petruzza, S., Stellato, A., & Turbati, A. (2010). Computer-aided Ontology Development: an integrated environment. *New Challenges for NLP Frameworks 2010 (held jointly with LREC2010).* La Valletta, Malta.

Parr, T. J., & Quong, R. W. (1994). ANTLR: A Predicated-LL(k) Parser Generator. *Software Practice and Experience , 25*, 789-810.

Petasis, G., Karkaletsis, V., Paliouras, G., Androutsopoulos, I., & Spyropoulos, C. D. (2002). Ellogon: A New Text Engineering Platform. *In Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC-2002).* Las Palmas, Canary Islands.