SODA
A Service Oriented Data Acquisition Framework

Andreea Dioșteanu[1], Armando Stellato[2], Andrea Turbati[2]
[1]*Bucharest Academy of Economic Studies, Romania;*
[2]*Tor Vergata University, Italy;*

ABSTRACT

In this chapter we present SODA (Service Oriented Data Acquisition), a service-deployable open-source platform for retrieving and dynamically aggregating information extraction and knowledge acquisition software components. The motivation in creating such a system came from the observed gap between the large availability of Information Analysis components for different frameworks (such as UIMA (Ferrucci & Lally, 2004) and GATE, (Cunningham, Maynard, Bontcheva, & Tablan, 2002)) and the difficulties in discovering, retrieving, integrating these components and embedding them into software systems for knowledge feeding. By analyzing the research area, we noticed that there are a few solutions for this problem though they all lack in assuring a great level of platform independence, collaboration, flexibility and, most of all, openness. The solution that we propose is targeted to different kind of users, from application developers, benefiting of a semantic repository of inter-connectable Information Extraction and Ontology Feeding components, to final users, which can plug&play these components through SODA compliant clients.

INTRODUCTION

While the Semantic Web (Berners-Lee, Hendler, & Lassila, 2001) is finally becoming a concrete reality, thanks to bootstrapping initiatives such as Linked Open Data (Bizer, Heath, & Berners-Lee, 2009) and assessment of W3C standards for expressing, querying and accessing distributed knowledge, still large part of the information available from the web is made available by traditional means: web pages and multimedia content.

To be able to cope with this huge volume of information, Information Extraction (IE) engines allow to lift relevant data from heterogeneous information sources and project it towards predefined knowledge schemes, thus enabling higher-level access based on semantic rather than textual indexing.

The purpose of such systems is actually two-fold: if documents (and media in general) are the focus, then these systems may support systems for document management, advanced semantic search, smart document tracking, etc. by identifying references to entities already available in knowledge bases, and indexing these documents with them; on the contrary, if the focus is on knowledge production, they may extract the information which is contained inside information sources, and compose it into semantic compound resources which can then be fed to knowledge bases.

The success of semantic search engines such as Eqentia[i] or Evri[ii] and Information Extraction and services such as OpenCalais[iii] and Zemanta[iv] show that there is large demand for this kind of solutions. However, all of them – while promising to break the old-fashioned concept of knowledge-silos by providing services and API for producing knowledge modeled according to open standards – still represent silos in their own offer, as users are not allowed to participate in the definition of new content lifters, nor they can access the code (or runnable instances) of the engines implicitly available through the provided services.

In this paper we present a novel framework which aims to overcome the above limitation, being completely based on standard technologies (such as UIMA for Unstructured Information Management) and models (from RDF to Web Ontology Language (W3C, 2004) and Simple Knowledge Organization Systems (W3C, 2009)) and offering an open architecture and platform for provisioning of IE components, which may help in composing systems for semi-automatic development and evolution of ontologies by lifting relevant data from unstructured information sources and projecting it over formal knowledge models.

BACKGROUND

Related Works

Applications such as the ones described above are becoming more and more popular because they provide lot of advantages like: facilitating communication between computers, creating standardized metadata, facilitate the management and processing of various document formats. However, the development of such applications requires lot of time and development effort.

Development-free solutions are based on services: this is the case of Thomson Reuters' OpenCalais. OpenCalais offers valid and robust services for tagging documents. However, the main drawbacks of its offer consist in the limitations of a completely web-based approach: it does not assure user document privacy[v] and also it does not offer any possible vertical customization to different domains. Under a certain point of view, we could say that Open-Calais is much more *close* than what its name would suggest.

Other approaches have been proposed in the scientific literature, describing less-opaque frameworks for composition of NLP components. In (Wilkinson, Vandervalk, & McCarthy, 2009), SADI – Semantic Automated Discovery and Integration - an open SOA framework for discovery of – and interoperability between – distributed data and analytical resources, is presented. The intent of SADI is very close to the one of OpenCalais, that is, to make service consumption of data analytics very simple: avoid the creation of further languages/technologies, remain tight to standard stateless GET/POST-based Web Services, combine them with W3C Semantic Web knowledge representation languages, and provide something in the middle between best-practices and an available implementation to prove its success. With respect to openness, SADI (as well as Open-Calais) services completely hides the details of the analytical components, though, this is just a requirement of its completely service-based architecture, which is focused on the description of the provided data rather than on the process to obtain it (thus allowing for any kind of analytical to be adopted and not constraining to any technology nor standard); however, differently from Open-Calais, SADI releases as open source the whole service framework (i.e. including the server and service orchestration).

Semantic Assistants (Witte & Gitzinger, 2009) provides web services specifications for consumption of data analysis (IE and IR) services. The services may be defined as pipelines of components based on the GATE (General Architecture for Text Engineering) architecture. A Semantic Assistants Ontology allows for a clear specification of actors in the service consumption, by defining the kind of *Artifact* which is being processed and its *Format*, the *User*, the *Language* (both describing the language of the document to be processed and the language understandable by the user), and the *Task* which is requested to be performed on the given Artifact. The authors also advocate a wider diffusion of desktop assistants to bring the power of text analytics in daily applications: their text engineering web service suite is thus complemented by dedicated client applications (Witte & Papadakis, 2009) developed as plug-ins for common tools such as email browsers and word processors.

A wider investigation on ontological descriptions for NLP architectures is provided in (Buyko, Chiarcos, & Pareja Lora, 2008), where a modular ontology describes various aspects of an NLP pipeline, with particular relevance given to the linkage between annotations from standard processing environments (such as UIMA) and ontological definitions from different linguistic vocabularies. A similar approach has also been introduced in (Cerbah & Daille, 2007), where a service-oriented architecture geared towards terminology acquisition is defined. The described architecture wraps NLP components as Web services with clearly specified interface definitions: language engineers can thus easily create and alter concatenations of such components.

Reference Frameworks

Probably, the two most quoted frameworks for NLP systems are the General Architecture for Text Engineering: GATE (Cunningham, 2002) and the Unstructured Information Management Architecture: UIMA[vi] (Ferrucci & Lally, 2004).

GATE was first released in 1996, inspired by the architecture for Text Engineering defined in the DARPA TIPSTER Project (Harman, 1992). Completely re-designed, rewritten, and re-released in 2002, the system is now one of the most widely-used systems of its type and is a relatively comprehensive infrastructure for language processing software development.

UIMA has originally been developed by IBM Research. In 2005, the US government sponsored the creation of the UIMA Working Group, a consortium of companies and universities committed to the exploration of UIMA as a framework for solving important NLP problems. As a result of the Working Group's activities, some existing NLP resources (such as Stanford's NLP library[vii] and OpenNLP[viii] toolkit) were integrated with UIMA. In early 2006, IBM published the UIMA source code on Source Forge and lately, in the same year, a new version of UIMA (Apache UIMA) had been made available as an open source project[ix] (hosted as an incubator) by the Apache Software Foundation, recently graduating for a position inside list of supported Apache projects, on 18th of March, 2010.

UIMA is also the name of an OASIS specification (Ferrucci, 2009) for a standard architecture for Unstructured Information Management; former UIMA is now known as UIMA Framework/Implementation or simply Apache UIMA.

Today, UIMA support can count on a wide open source community made up of developers (contributing to the main project) and users (in turn, developers too, which however contribute with NLP software released under UIMA specifications). Conversely, GATE benefits of 15 years of maturity, of a plethora of available software (free open-source as well as commercial off-the-shelf products) compatible with its architecture which has been written in these years, and on the availability of analysis and manipulation components, such as the JAPE transducer (Java Annotation Patterns Engine), able to manage elements extracted over annotations basing on regular expressions.

Other frameworks which have close commonalities with our environment are those related to software provisioning and dependency resolution, such as Apache Ivy[x], Maven[xi] or Orbit[xii].

Orbit is a project developed inside the Eclipse community to provide a repository of bundled versions of third party libraries that are approved for use in one or more Eclipse projects. Ivy is focused on dependency management of software components (from libraries to elaborated components or DB resources) inside the popular build system Apache ANT[xiii] while Maven is the current standard-de-facto for software description and provisioning, featuring a rich model for the describing software projects (the POM: Project Object Model) and a powerful dependency resolution mechanism.

Component description and dependency resolution (in this case, component-component as well as task-component and task-task are being considered) are two key features of the presented work too.

Ontology Development, Learning and Evolution Frameworks

Nowadays, basic architectural definitions and interaction modalities have been defined in detail fulfilling industry-standard level for processes such as:

- ontology development with most recent ontology development tools following the path laid by Protégé (Gennari, et al., 2003)
- text analysis (see previous section).

on the contrary a comprehensive study and synthesis of an architecture for supporting ontology development driven by knowledge acquired from external resources, has not been formalized until now.

What lacks in all current approaches is an overall perspective on the task and a proposal for an architecture providing instruments for supporting the entire flow of information (from acquisition of knowledge from external resources to its exploitation) to enrich and augment ontology content. Just scoping to ontology learning, OntoLearn (Velardi, Navigli, Cucchiarelli, & Neri, 2005) provides a methodology, algorithms and a system for performing different ontology learning tasks, OntoLT (Buitelaar, Olejnik, & Sintek, 2004) provides a ready-to-use Protégé plugin for adding new ontology resources extracted from text, while the sole Text2Onto (Cimiano & Völker, 2005) embodies a first attempt to realize an open architecture for management of ontology learning processes.
If we consider ontology-lexicon integration, previous studies dealt with how to represent this integrated information (Peters, Montiel-Ponsoda, Aguado de Cea, & Gómez-Pérez, 2007; Buitelaar, et al., 2006; Cimiano, Haase, Herold, Mantel, & Buitelaar, 2007), other have shown useful applications exploiting onto-lexical resources (Basili, Vindigni, & Zanzotto, 2003; Peter, Sack, & Beckstein, 2006) though only few works (Pazienza, Stellato, & Turbati, 2008) dealt with comprehensive framework for classifying, supporting, testing and evaluating processes for integration of content from lexical resources with ontological knowledge.

MOTIVATION

In our exploration of current state-of-the-art on semi-automatic development of structured data, we shared the same perspective and considerations expressed in (Witte & Gitzinger, 2009): "While more advanced semantically-oriented analysis techniques have been developed in recent years, they have not yet found their way into commonly used desktop clients, be they generic (e.g. word processors, email clients) or domain-specific (e.g., software IDEs, biological tools)."
Also, we agree with their perspectives on ideal directions for future development: "Instead of forcing the user to leave his current context and use .. external application(s) …", it is way better to leave users adopt their usual applications, possibly enriched with extensions and plugins for invoking "analysis services relevant for [their] current task". However, another consideration drove our research effort: current hit&consume services provide very simple and no-customizable-at-all data acquisition from documents, whereas the full potential of natural language processing could demand for a wider support on the web, embracing final users as well as developers (sitting on various and different steps of the development stairway…) in a more comprehensive and global experience. In this sense, we foresee the existence of environments for component provisioning and composition, much in the spirit of Maven and Ivy (and possibly sitting on top of them), but targeted towards the more specific area of Unstructured Information Management and Data Acquisition. Components could then be easily downloaded and aggregated by developers through dedicated IDEs exploiting the provisioning services, or be immediately used by final users through dedicated and compliant service-consumers embedded in desktop applications as well.

Objectives

The solution that we are addressing should cover the motivations expressed above, by pursuing the following objectives:

1. adopting a standard for Unstructured Information Management (UIM)
2. providing (or adopting) a standard for cataloguing UIM components
3. providing a standard for expressing the necessary transformations to project semantically annotated information into the desired data structures (e.g. projecting text annotations over the data structures of a given ontology)
4. enabling data-driven semi-automatic composition of UIM and Data Projection components (e.g. the user defines the ontology for which they want to produce data, and the system is able to define a set of components properly composed – they may operate in a chain, or simply sum-up their different kind of extraction capabilities, or a combination of both – to extract data from unstructured information sources and project it towards the given ontology).
5. provisioning components according to the composition modalities defined above

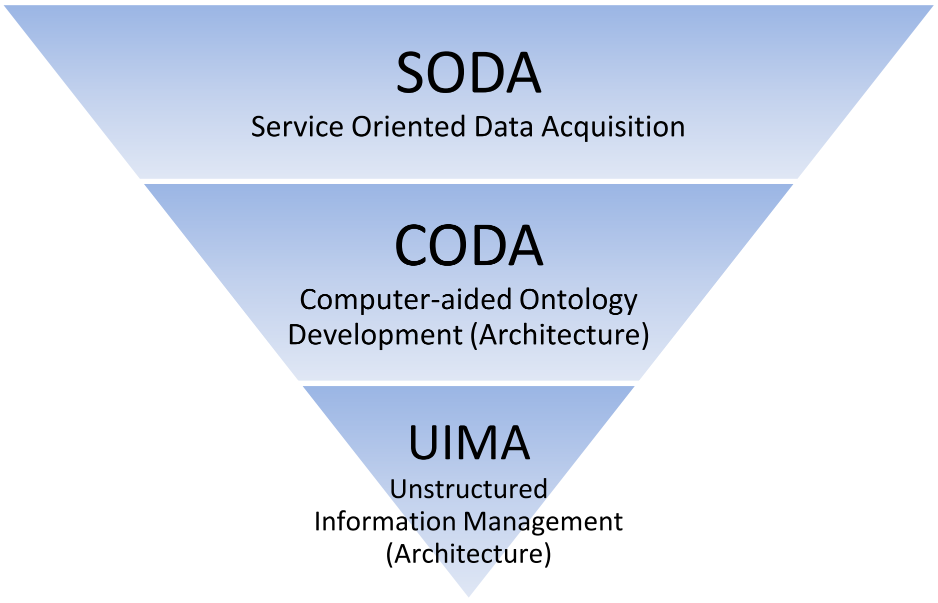

SODA
Service Oriented Data Acquisition
CODA
Computer-aided Ontology Development (Architecture)
UIMA
Unstructured Information Management (Architecture)

Figure 1 SODA Technology Stack

THE SODA ARCHITECTURE AND FRAMEWORK

Considering the objectives above, we found there is need for plenty of models, formats and technologies going from the analysis of unstructured information up to the acquisition of knowledge for structured data sources, and that these would better be organized in separate layers rather than in an intricate forest of specifications (or, even worse, in a underspecified technology melting pot).

Thus, as a first action, we laid up a technology layer cake addressing our requirements and providing independent levels of application (i.e. starting bottom-up, each layer may be considered as a contribution to information management which is complete in its objectives and does not need to be motivated by its role in higher layers). The layer-cake, which is reported in Figure 1, is composed of:

1. *Unstructured Information Management*: supports creation, integration and deployment of unstructured information management solutions from combinations of semantic analysis and search components. Comprehends, but is not limited to, text analysis and understanding, audio and video processing etc… with the intent of providing structured chunks of information extracted from the unstructured information streams, which can then be searched, reused, transformed etc… For this layer, we decided to stand on top of mature results available from state-of-the-art technologies in UIM, and have chosen the OASIS standard UIMA embracing its information models (e.g. Feature Structures, Type Systems) and the whole software architecture and framework as well.

2. *Computer-Aided Ontology Development*: this layer deals with the enrichment and evolution of ontology content through exploitation of unstructured information sources, by using (semi)automatic approaches. While the first layer provides resources analysis and sensible information extraction, the specifications in this layer allow for the reuse and transformation of the above extracted information to populate ontologies[xiv]. For this layer, we have worked on the definition of an Architecture, CODA: Computer-Aided Ontology Development Architecture (Fiorelli, Pazienza, Petruzza, Stellato, & Turbati, 2010) and in the development of an associated Software Framework. The CODA Framework completely delegates and exploits UIMA for what concerns Unstructured Information Analysis and Extraction; it then provides a dedicated language

for projecting UIMA Annotations (or, in general, instances of UIMA Type Systems) over RDF repositories and a series of components for supporting this task.

3. *Service Oriented Data Acquisition:* while the second layer provides all the details for building systems and components for semi-automatic development of ontologies, the third layer focuses on the provisioning of such components. Based on the Maven-like approach discussed in previous sections, but tailored versus CODA, the Service Oriented Data Acquisition Framework establishes services and specifications for searching, retrieving and plugging CODA components. The search specification should be considered as simple as possible, completely abstracting from the details of the needed components and only focusing on the knowledge need (e.g. the ontology/thesaurus/conceptual scheme for which the information need to be provided) and on the context of the subject of analysis (e.g. the processed media, their format etc…). The SODA provisioning system can also be seen as a developer's open library, providing a component repository to be inspected and consumed by development environments, where users can search, download and compose components to rearrange new analysis engines, possibly contributing with their own developed components, again, much in the spirit of similar solutions for library provisioning, such as Maven (Dodinet, 2005)

In the following sections we first describe the CODA architecture and framework for supporting Semi-Automatic Development of Ontologies, and consequently move to the following layer, showing how, through SODA, CODA components can be easily searched, retrieved and plugged to running systems for Knowledge Processing and Acquisition.

CODA

The middle tier of the SODA technology layer cake is CODA (Fiorelli, Pazienza, Petruzza, Stellato, & Turbati, 2010), the *Computer-aided Ontology Development Architecture*. Whereas the SODA specification deals with remote publication and retrieval of components, CODA acts as a core backbone of the SODA distributed framework and is the interlingua that all SODA compliant client machines must be able to understand.

Though the objectives of the full CODA+SODA framework have already been provided, we define here the specific expression "Computer-aided Ontology Development" as to include all processes for enriching ontology content through exploitation of external resources, by using (semi)automatic approaches.

COD tasks cover:

1. **(Traditional) Ontology Learning** tasks, devoted to augmentation of ontology content through discovery of new resources and axioms. These include discovery of new concepts, concept inheritance relations, concept instantiation, properties/relations, domain and range restrictions, mereological relations, equivalence relations etc…
2. **Population of ontologies with new data**: a rib of the above, this focuses on the extraction of new ground data for a given (ontology) model (or even for specific concepts belonging to it)
3. **Linguistic enrichment of ontologies**: enrichment of ontological content with linguistic information coming from external resources (eg. text, linguistic resources etc…)

As for UIMA, CODA is the given name of both an architecture and of its associated platform, for this reason we use the whole name CODA even when referring to its architecture (whereas the final "A" stands for "Architecture") instead of just COD.

CODA Architecture

The Architecture of CODA defines the components and their interaction at an abstract level, which are then implemented in the homonymous framework. This architecture builds on top of the industry standard UIMA (tasks 1&2) and, for task 3, on the Linguistic Watermark (Pazienza, Stellato, & Turbati, 2008) suite of ontology vocabularies and software libraries for describing linguistic resources and the linguistic aspects of ontologies. Figure 2 depicts the part of the architecture supporting tasks 1 and 2. Tiny arrows

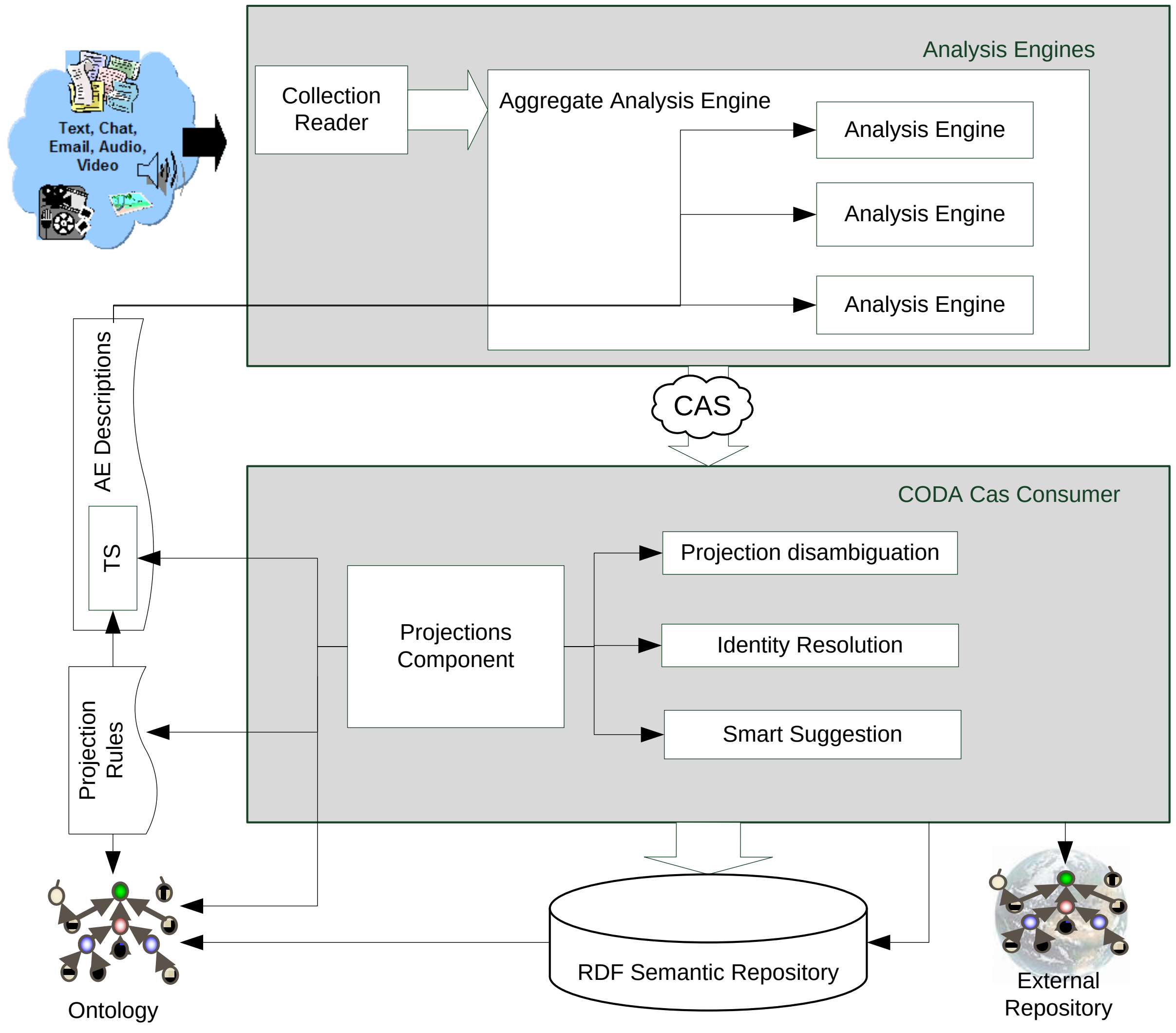


Figure 2 CODA Architecture

represent the *use/depends on* relationship, so that the Semantic Repository (bottom part of the figure) owl:imports the reference ontologies, the projection component *invokes* services from the other three components in the CODA CAS Consumer as well as *is driven by* the projection document and Type System (TS) and reference ontology. Large arrows represent instead the flow of information.
While UIMA already foresees the presence of CAS Consumers (see CAS Consumer entry on the terms glossary at the end of this chapter) for projecting collected data over any kind of repository (ontologies, databases, indices etc…), COD Architecture expands this concept by providing ground anchors for engineering ontology enrichment tasks, decoupling the several processing steps which characterize development and evolution of ontologies. This is our main original contribution to the framework.
Here follows a description of the presented components.

Projection Component (Core of the CODA Platform)

This is the main component which realizes the projection of information extracted through traditional UIM components (i.e. UIMA *Annotations*).

The Unstructured Information Management (UIM) standard foresees data structures stored in a CAS (Common Analysis System). CAS data comprises a *type system*, i.e. a description – represented through feature structures (Carpenter, 1992) – of the kind of entities that can be manipulated in the CAS, and the *data* (modeled after the above type system) which is produced over processed information stream (see also the term glossary at the end of this chapter).
This component thus takes as input:

- A Type System (TS from now on)
- A reference *ontology* (we assume the ontology to be written in the RDFS or OWL W3C standard)
- A projection document containing projection rules from the TS to the ontology
- A CAS containing annotation data represented according to the above TS

and uses all the above in order to project UIMA annotations as data over a given Ontology Repository.
A projection rule specification (see section on *PEARL* language in the next page for details) has also been established, providing a flexible language for projecting arbitrary extracted data into the target ontology.
The Projection Component can be used in different scenarios (from massively automated ontology learning/population scenarios, to support in human centered processes for ontology modeling/data entry) and its projection processes can be supported by the following components.

Projection Disambiguation Component(s)

These components may be invoked by the Projection Engine to disambiguate between different possible projections. Projection documents may in fact describe more than one projection rule which can be applied to given types in the TS. These components are thus, by definition, associated to entries in Projection Documents and are automatically invoked when more than a rule is matched on the incoming CAS data.
This component has access by default to the managed Semantic Repository (and any reference ontology for the Projection rules), to obtain a picture of the ongoing process which can contribute to the disambiguation process.

Identity Resolution

Whenever an annotation is projected towards ontology data, the services of this component are invoked to identify potential matches between the annotated info which is being reified into the semantic repository, and resources already present inside it.
If the Identity Resolution (IR) component discovers a match, then the new entry is suggested to be the same as the pre-existing one; that is, any new data is added to the resource description while duplicated information (probably the one which helped in finding the match) is discarded.
The IR component may look up on the same repository which is being fed by CODA though also external repositories of LOD (linked open data) can be accessed. Eventually, entity naming resolution provided by external services – such as the Entity Naming System (ENS) OKKAM (Bouquet, Stoermer, & Bazzanella, 2008) – may be combined with internal lookup on the local repository. The component interface is agnostic with respect to the algorithms used to compute the identity, and are thought instead for properly informing the system whenever matching identities are found during resource generation.
The Input for this component is composed of:

- External RDF repositories (providing at least search functionalities over their resources)
- Entity Naming Systems access methods
- Other parameters needed by specific implementation of the component

Smart Suggestion Component(s)

These components help in proposing suggestions on how to fill empty slots in projection rules (such as subjects in datatype property projections or free variables in complex FS to graph-pattern projections). As for Disambiguation Components, these components can be written for specific Projection Documents and associated to the rules described inside them, as supporting computational objects.

```
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 ('=' val=parameterValue)? ;
```

Figure 3 BNF of (part of) the grammar used in the Projection Rule File

PEARL: the ProjEction of Annotations Rule Language

One of the main features of CODA is PEARL, the ProjEction of Annotations Rule Language: a language used to describe projections from annotations taken after UIMA Type Systems towards RDF. A simplified version of the language grammar, expressed in Backus-Naur form, can be seen in Figure 3.
Core elements of the language are the Projection Rules, which enable users to describe matches over a set of annotations taken by UIMA components over streams of unstructured information, and to specify how the matched annotations will be transformed into sensible information which will be in turn imported inside the target ontology. The concept is very close to XSL Transformations over XML: template + rewrite rules. The main difference resides in its applicability of its templates at model level (RDF) instead of syntactic level (XML syntactic patterns).
We describe here the structure of a typical projection document, namely a document containing a set of projection rules.

Structure of a Projection Document

Each projection document (a document containing PEARL rules) can be considered as divided into two main parts: in the upper part there is a listing of all the namespace prefixes that will be used in the projection rules for what concerns ontology resources, and the second part, which is normally longer, contains the projections rules. An example of a Projection Document with two rule declarations is provided in Figure 4.

```
my=http://Publication#;
xsd=http://www.w3.org/2001/XMLSchema#;
foaf=http://xmlns.com/foaf/0.1/;
bibo= http://purl.org/ontology/bibo/;


rule it.uniroma2.Book id:book1 0.9 {

      nodes = {
            book        uri                       _it.uniroma2.Book:title
            title       plainLiteral              _it.uniroma2.Book:title
            author      uri                       _it.uniroma2.Book:author
            authorName  literal(xsd:String)       _it.uniroma2.Book:author
            editor      uri                       _it.uniroma2.Book:editor
            isbn        literal(xsd:String)       _it.uniroma2.Book:isbn
      }

      graph= {
            $book       a                 bibo:Book     .
            $book       <bibo:title>      $title        .
            $book       bibo:author       ?author       .
            ?author     foaf:name         $authorName .
            $book       bibo:editor       $editor       .
            $book       bibo:isbn         $isbn         .
      }

      Where = {
            ?author     foaf:name         $authorName .
      }

      parameters={mandatory,reference=false};

}


rule it.uniroma2.Site dependsOn last(book1) {

      nodes={
            site        literal(xsd:anyURI)       it.uniroma2.Site
      }

      graph = {
            $book1:book my:site           $site
      }
}
```

Figure 4 Example of a Projection Rule File

Prefix Declaration

The first part of a projection document contains all the ontology prefixes (my, xsd, bibo and foaf in the example) 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, are used to expand prefixed names inside the document into valid RDF URIs and no trace of them is left in the target ontology.

Projection Rules

After the prefix declaration, the projection document lists the projection rules and their descriptions (2 projection rules in the example).

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* and *parameters*.

Rule Declaration

Each rule starts with a declaration, expressed through the keyword “rule” and concluding with a curly bracket "{" initiating its definition. The first element in the declaration is the UIMA type from the adopted UIMA Type System: any UIMA annotation taken after that type (written following the UIMA standards regarding types, as a dot-separated package name followed by a capitalized word referring to the Type in the UIMA Type System) will trigger the rule. After the type declaration, there is an optional rule identifier that can be used to make references to a rule from other rules, according to different relationships of dependency. Then we have a number in the range of 0..1, representing a confidence value which can be used by to rank different rules of the same type. Decision whether to apply all rules, only the first one or, for example, letting the users make their choice, is up to the external application exploiting CODA. The declaration may end with a list of dependencies to other rules. Each dependency must specify the *relationship* which is being established (e.g. the second rule in the example in figure 4 states that when this rule will be used it will have a dependency with the *last* occurrence of the first rule).

Alias

This optional section is used to define aliases that will be used in the Nodes part (see next section). An alias is a compact way to use a value which is present inside a feature of the given annotation type. This means that by using an alias ('$'+alias_name) in the Nodes part we mean the value, if present, contained in that particular feature.

Nodes

The third part, which is optional only if the rules depends on another one, provides a list of placeholders for ontology nodes. 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. For instance, in the first rule in figure 4, the UIMA feature *it.uniroma2.Book:title* will be projected both as an RDF named resource (URI) and as a simple string (plainLiteral): in the first case an URI is created after the feature’s content (book placeholder), and will be used as the resource identifying the extracted book, while in the second case a literal value (title placeholder) is generated to populate a property of the book (see next *Graph* section for details about the projection). A set of operators are available for applying different transformations to the features, converting them into valid RDF nodes. Default conversions are applied when no operator is specified, and are inferred on the basis of the specified 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).
In UIMA it is possible for a value of a particular feature to be a type itself, thus containing other features and so on recursively, as stated in the feature structure theory: *Feature Path* is a standard notation introduced in UIMA to identify arbitrary values in complex feature structures by describing navigation of nodes into up to the desired value, much in the spirit of XPath for XML.
Sometimes it may be necessary to assign one feature (its value) or statically an RDF resource (e.g. a given OWL Class/Property) depending on the value of another feature. This can be done using the *alias* mechanism and a simple if/else construct as explained in the grammar (the alias is used in the "condBool”). So for example it is possible to assign to the placeholder 'gender' the OWL class 'Male' or 'Female' by checking a value of a feature (if that feature has the value 'mr.' we may assign 'Male', conversely, 'Female' will be assigned for the value 'mrs.'). This can be useful because in the GRAPH part of a rule (see next section on graphs) we can have the triple '$person a $gender', which is more specific than '$person a foaf:Person'

Graph

The graph section contains the true projection over the target ontology graph, by describing a graph pattern which is dynamically populated with grounded placeholders and variables (see next paragraph on the where section). The graph pattern[xv] consists in 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) are identified by the prefixed symbol "$". Getting back to the example discussed in the previous paragraph, the book placeholder is used in five different triples, always as the subject, since it is the main resource which is being described through other features extracted with UIMA, the title placeholder, which originated from the same feature (but has been converted in a different way, providing a human readable label instead of a formal URI), is instead used to populate the bibo:title property linked to the previous URI. RDF nodes can be references in graph patterns through the usual notation for URIs ("< >" delimited standard URIs) or by prefixed local names. The abbreviations are represented by a finite list of words that can be used in place of explicit reference to RDF resources. For example we included in this list the standard abbreviation from the RDF Turtle format (Beckett & Berners-Lee, 2008) – also adopted in the SPARQL query language – which assumes the character "a" to be interpreted as rdf:type.
Finally, it is possible to use *variables* (by prefixing their names with a "?" symbol) when there is a 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 paragraph for more details).

Where

As for *graph* section, the *where* section contains a graph pattern: this pattern is matched over the target ontology to retrieve nodes already existing in it by means of variable unification (variables are identifiable by a prefixed "?" symbol), 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 much close to the purpose of the *where* statement in a SPARQL CONSTRUCT[xvi] 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. New operators can be introduced by custom Identity Resolution components (PEARL is extensible much in the same way as SPARQL), to search values over the target ontology (or external data such in the case of Entity Naming Systems) instead of just unifying them.

Parameters

The fifth and last part, optional, consist 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 CODA component implementations which may be 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.

Rule Execution

The execution flow of a rule is shown in Figure 5: the depicted status of placeholders and variables is considered to represent their assignment at the start of each phase. The execution starts by parsing and replacing all prefixes and assigning them to namespaces, then it moves to the *nodes* section, by creating values for the placeholders; these values are then assigned to placeholders appearing in the graph of the *where* section (and forwarded to the *graph* too), which is then matched over the target ontology graph, by unifying the variables contained into it. Ground values assigned to variables after the graph match and the

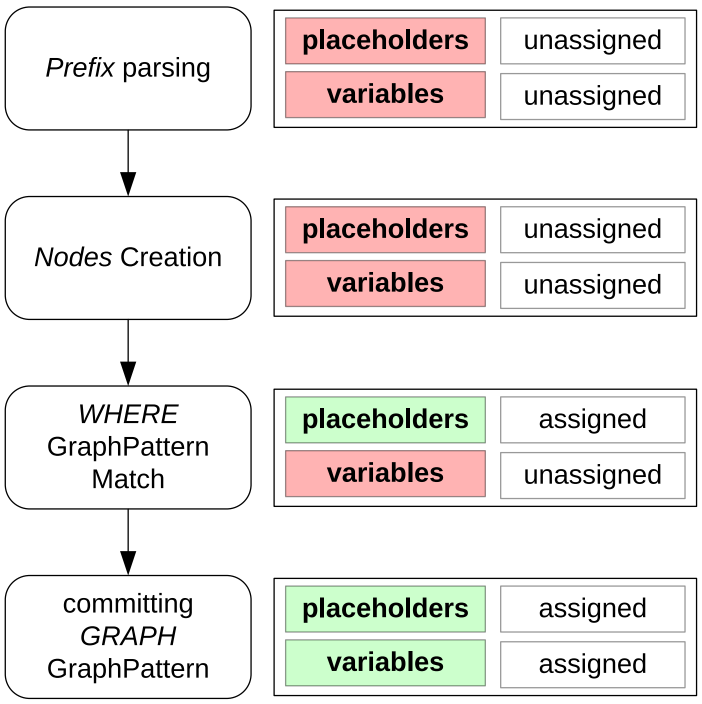


Figure 5 PEARL Execution Flow

already calculated placeholders are then replaced in the graph pattern of the *graph* section, the content of which is finally added to the target ontology.
Both the *WHERE* and *GRAPH* graph patterns applications can fail. In the first case, the *WHERE* graph application is a match against the target ontology: a failed match is always considered as having a result-set of a single tuple with all variables unbound. All of SPARQL operators can be used in the *WHERE* clause to alter the matching strategy.
The application of the *GRAPH* graph pattern (that is, writing triples to the target ontology) is different: 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.
If the application of a *GRAPH* graph pattern write fails, no triple is written to the target ontology (for that rule application).
It is possible to note how, in the example in figure 4, the placeholder $author is never used in the rule, though it is being created inside the *nodes* rule fragment. This brings in another feature of the variable assignment process: whenever a variable in the final *GRAPH* pattern is not bound, before declaring that write to be failed, the processor tries to match it with a placeholder of identical name. In the case of the example, the variable ?author is used two times: the first time as the object of the property bibo:author and the second time as the subject of the property foaf:name. The value of that variable is bound to a match in the target ontology, looking for already existing authors with the same name: if the match succeeds, then the variable is bound and initialized with the URI already assigned to the author with that name, conversely, if the match fails, the variable is not bound, but a placeholder with the same name is found, so its value will be used in place. The idea is that if the author already exists in the target ontology, then its URI is retrieved, otherwise a URI is generated for it.

Templates

Templates are a useful feature of the PEARL language which allow, very closely to programming languages such as C++, to create parametric definitions of rules which can be saved into libraries of projections and then be easily identifiable and reusable in different contexts. A template projection rule can thus be identified by a simple *template signature*, which is composed of the template name and by its set of parameters (which occur in the template rule definition). New rules can then be generated by referring the opportune template and by assigning values to the parameters. Rule resolution does not change, it is handled as a pre-processer macro expansion: simply a projection document containing reference to templates is rewritten by expanding the template references in their rule definitions, by replacing the parameters with their assigned values, and then the normal PEARL execution flow is activated.
Here we provide two simple examples of rule templates:

- *Projecting CAS feature structures (FS) as instances of a given class*. A single entity from a UIMA type system can be projected as an instance of a given class
- *Projecting FSs as values of datatype properties*. This requires ontology instances to be elected as subjects for each occurrence of this projection, usually by specifying a reference to another rule.

Another relevant feature of rule templates is the possibility to specify *decorators* for parameters: decorators may be used to convey semantics to external applications about the nature of the objects to be projected. For instance, an interactive annotation tool recognizing a given parameter to be recognized as a *class* resource, would allow the user – anytime a given template rule implementation is being applied – to choose the proper class among the subclasses of the one assigned to the parameter, by showing a *class tree* rooted on it.

The CODA Framework and its objectives

CODA Framework is an effort to facilitate development of systems implementing the COD Architecture, by providing a core platform and highly reusable components for realization of COD tasks.
Main objectives of this architectural framework are:

1. Orchestration of all processes supporting COD tasks
2. Interface-driven development of COD components
3. Maximizing reuse of components and code
4. Tight integration with available environments, such as UIMA for management of unstructured information from external resources (e.g. text documents) and Linguistic Watermark (Pazienza, Stellato, & Turbati, 2008) for management of linguistic resources
5. Minimizing required LOCs (lines of code) and effort for <u>specific COD component development</u>, by providing high level languages (and implementations for associated processors) for matching/mapping components I/O specifications instead of developing software adapters for their interconnection
6. Providing standard implementations for components realizing typical support steps for COD tasks, such as management of corpora, user interaction, validation, evaluation, production of reference data (oracles, gold standards) for evaluation, identity discoverers etc…

With respect to components described in the previous section, the CODA Framework provides the main Projection Component (and its associated projection language), a basic implementation of an Identity Resolution Component, and all the required business logic to fulfill COD tasks through orchestration of COD components.

Possible application scenarios

In our attempt to fulfill the above objectives, we,envision several application scenarios for CODA. We provide here a description of a few of them.

Fast Integration of existing UIMA components for ontology population

By providing projections from CAS type systems to ontology vocabularies via PEARL, one could easily embed standard UIMA AEs (Analysis Engines) and make them able to populate ontology concepts pointed by the projections, without requiring development of any new software component. Moreover (objective 6 above), standard or customized identity discoverers will try to suggest potential matches between entities annotated by the AE and already existing resources in the target ontology, to keep identity of individual resources and add further description to them. In this scenario, given an ontology and a AE, only the projection from the CAS type system of the AE to the ontology is needed (and optionally, a customized identity discoverer). Everything else is assumed to be automatically embedded and coordinated by the framework.

Rapid prototyping of Ontology Learning Algorithms

This is the opposite situation of the scenario above. CODA, by reusing the same chaining of UIMA components, ontologies, CAS-to-Ontology projections, identity discoverers etc… , may provide:

- a preconfigured CAS type system (Ontology Learning CAS Type System) for representing information to be extracted under the scope of standard ontology learning tasks
- preconfigured projections from above CAS type system to learned ontology triples, in the form of PEARL templates (see subsection on PEARL templates) dedicated to Ontology Learning
- extended interface definitions for UIMA analysis engines dedicated to ontology learning tasks: available abstract adapter classes will implement the standard UIMA AnalysisComponent interface, interacting with the above Ontology Learning CAS type system and exposing specific interface methods for the different learning tasks

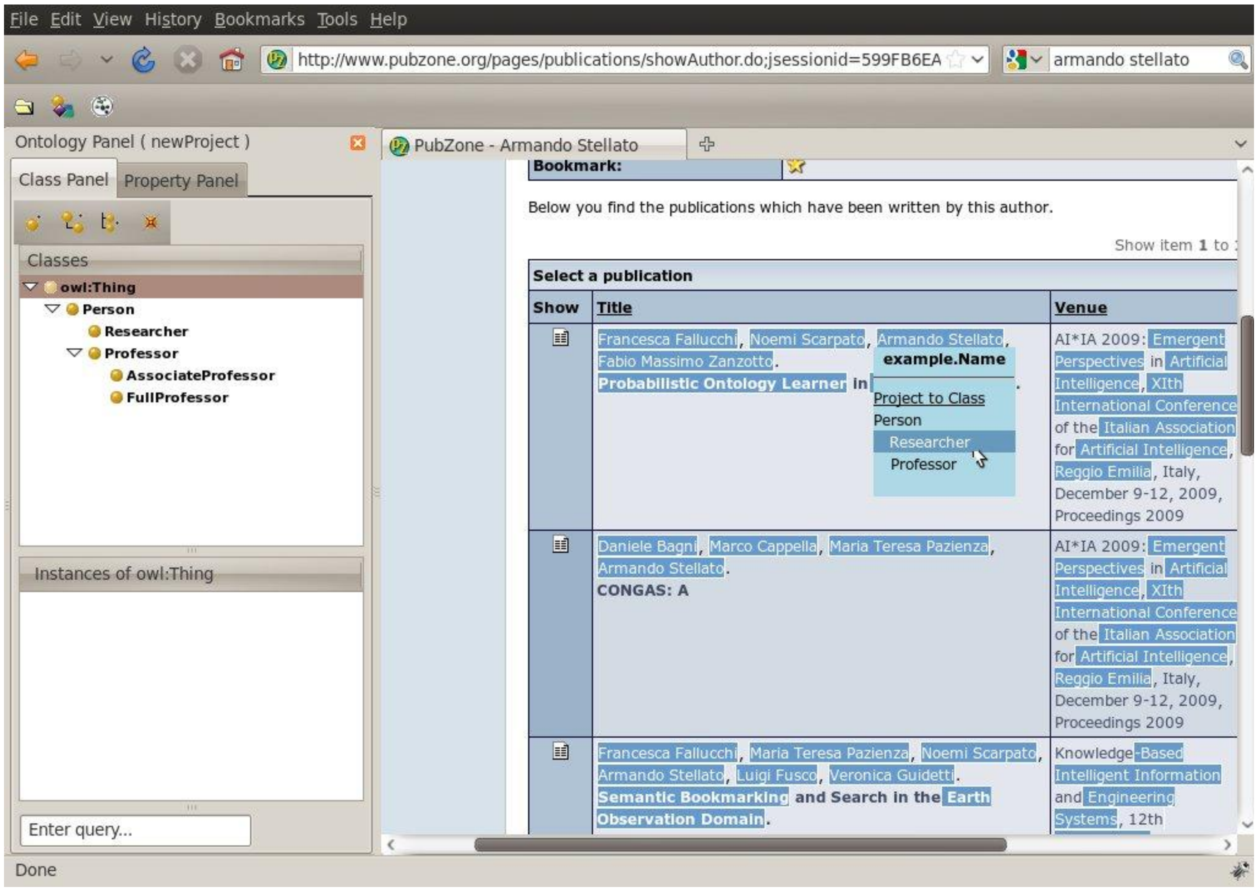
In this scenario, developers willing to rapidly deploy prototypes for new ontology learning algorithms, will be able to focus on algorithm implementation and benefit of the whole framework, disburdening them from corpora management and generation of ontology data. This level of abstraction far overtakes the *Modeling Primitive Library* of Text2Onto (i.e. a set of generic modeling primitives abstracting from specific ontology model adopted and being based on the assumption that the ontology exposes at least a traditional object oriented design, such as that of OKBC (Chaudhri, Farquhar, Fikes, Karp, & Rice, 1998)). In fact in CODA ontology learning tasks are actively defined in terms of reusable projection rules, which just provide UIMA anchors for Analysis Engines to be developed by researchers. For instance, pairs of terms could be produced by taxonomy learners, which are then be projected as IS-A or type-of relationships by the framework.

Plugging algorithms for automatic linguistic enrichment of ontologies

In such a scenario, the user is interested in enriching ontologies with linguistic content originated from external lexical resources. The Linguistic Watermark library - which is already been used in tools for (multilingual) linguistic enrichment of ontologies (Pazienza, Stellato, & Turbati, 2010) and which constitutes a fundamental module of CODA - supports uniform access to heterogeneous resources wrapped upon a common model for lexical resource definition, allows for their integration with ontologies and for evaluation of the acquired information. Once more, the objective is to relieve developers from technical details such as resource access, ontology interaction and update, by providing standard facilities associated to tasks for ontology-lexicon integration/enrichment, and thus leaving up to them the sole objective of implementing enrichment algorithms.

User Interaction for Knowledge Acquisition and Validation

User interaction is a fundamental aspect when dealing with decision-support systems. Prompting the user with compact and easy-to-analyze reports on the application of automated processes, and putting at his hands instruments for validating choices made by the system can dramatically improve the outcome of processes for knowledge acquisition as well as support supervised training of these same processes. CODA front-end tools should thus provide CODA specific applications supporting training of learning-based COD components, automatic acquisition of information from web pages visualized through the



File Edit View History Bookmarks Tools Help

http://www.pubzone.org/pages/publications/showAuthor.do;jsessionid=599FB6EA

armando stellato

Ontology Panel (newProject)

Class Panel | Property Panel

Classes

- owl:Thing
  - Person
    - Researcher
    - Professor
      - AssociateProfessor
      - FullProfessor

Instances of owl:Thing

Enter query...

PubZone - Armando Stellato

Bookmark:

Below you find the publications which have been written by this author.

Show item 1 to

Select a publication

Show	Title	Venue
	Francesca Fallucchi, Noemi Scarpato, Armando Stellato, Fabio Massimo Zanzotto. **Probabilistic Ontology Learner** in	AI*IA 2009: Emergent Perspectives in Artificial Intelligence, XIth International Conference of the Italian Association for Artificial Intelligence, Reggio Emilia, Italy, December 9-12, 2009, Proceedings 2009
	Daniele Bagni, Marco Cappella, Maria Teresa Pazienza, Armando Stellato. **CONGAS: A**	AI*IA 2009: Emergent Perspectives in Artificial Intelligence, XIth International Conference of the Italian Association for Artificial Intelligence, Reggio Emilia, Italy, December 9-12, 2009, Proceedings 2009
	Francesca Fallucchi, Maria Teresa Pazienza, Noemi Scarpato, Armando Stellato, Luigi Fusco, Veronica Guidetti. **Semantic Bookmarking and Search in the Earth Observation Domain**.	Knowledge-Based Intelligent Information and Engineering Systems, 12th

Done

Figure 6. Knowledge Acquisition with UIMAST

browser (or management of info previously extracted from entire corpora of documents) and editing of main CODA data structures (such as UIMA CAS types, projection documents and, obviously, ontologies). Interactive tools should support iterative refinement of massive production of ontology data as well as human-centered process for ontology development/evolution.

This last important environment is a further very relevant objective, and motivated us to develop UIMAST (Fiorelli, Pazienza, Petruzza, Stellato, & Turbati, 2010), an extension for Semantic Turkey (Griesi, Pazienza, & Stellato, 2007) – a Semantic Web Knowledge Acquisition and Management platform[xvii] hosted on the Firefox Web Browser – to act as a CODA front-end for doing interactive knowledge acquisition from web pages. Figure 6 displays a screenshot from UIMAST, where a standard PEARL template for populating classes with instances is applied to a UIMA Named Entity Recognizer: UIMAST prompts the user with a cascade of classes rooted on the class which is being mapped to the Named Entiry Recognizer (Person), taken from the currently edited ontology. In this scenario, the user has yet full control of the Knowledge Acquisition process, yet their effort is incredibly lessened as their choices are strongly driven by constraints of the imported projection rules.

SODA

The upper layer of the whole SODA layer cake (called SODA itself) is the true part dedicated to Service-Oriented Data Acquisition, so it is focused on an open architecture and framework for provisioning of CODA components. In short, with respect to available extraction-on-demand services, SODA provides the required components that a SODA compliant client can assemble to semi-automatically compose data acquisition systems. This process, as already discussed, has a series of advantages, such as:

1. *total data protection guaranteed at physical level*: unstructured information is processed on the client machine and is never sent to the service. There is no need of any security infrastructure nor of trusting the SODA provider[xviii]
2. *support for very large corpora and heavy-processing*: again, thanks to the downloaded components being activated on the client machine, users are free to setup heavy-processing hardware and just get the business logic (CODA components) from the web. This does not prevent the creation of OpenCalais-like solutions based on SODA, i.e. where the service takes care of all the processing, acquires the unstructured information and returns result to the client. Simply, the system is truly open, and all options in this sense are viable.
3. *an increased level of flexibility*. SODA is a provider of CODA/UIMA components and of CODA projection documents as well. The SODA search services can be used to download the components and projections which mostly reflect user needs, and customize them to exactly fit these needs, for instance, by adapting projections for a given "famous" vocabulary, to a specific vocabulary which is not supported by SODA or by replacing one UIMA component with a locally improved version (by using the same UIMA Type System or by adapting the projections to the one which is being adopted by the replacing component)

The main services provided by SODA are:

- *retrieval of CODA projection documents* for a given vocabulary specified by the user
- *download of UIMA components*, by inspecting available UIMA AEs and matching them with those required by available projection documents for the specified ontology

Further services are then built around this platform, still with the intention of satisfying specific needs and exigencies related to the provisioning of data acquisition components. For instance, whenever no projection document is available for a given vocabulary, the system provides a semantic matching service which looks for projection documents available for ontologies having a domain overlap with the desired one. If one or more candidates are found, a new projection document is dynamically produced by properly readapting those which have been retrieved, to project towards the desired target vocabulary. Results of the above process are also cached, and can be proposed to the next user requesting CODA setups for the same vocabularies, while specifying that they have been automatically produced (for purposes of assessing quality of extraction components).

SODA Architecture

SODA is flexible and scalable service oriented platform-independent solution. The functionalities of the platform may be extended by integrating further components, such as new web services or API calls.
At the time being SODA Architecture is composed of the modules described in Figure 7. The following sections provide detailed descriptions for all of them.

Main Web Service Module

The scenario for consuming this service is the following one: a SODA enabled client is looking for appropriate Information Extraction components to populate the ontology it is being managing (we refer to it as the *target ontology* from now on) and thus sends a request to this service, specifying the namespace of the target ontology.
The role of this web service is thus to establish communication with the client, that is to register SODA enabled clients' requests (in the form of the namespace of their target ontology) and to return to them the UIMA Pear Component and the Projection Rule Document which are most appropriate for the client's target ontology. The output containing all the previously mentioned components is returned to the client either immediately or after required offline processing (such as the above mentioned semantic matching) has been concluded, by means of a ticket mechanism.

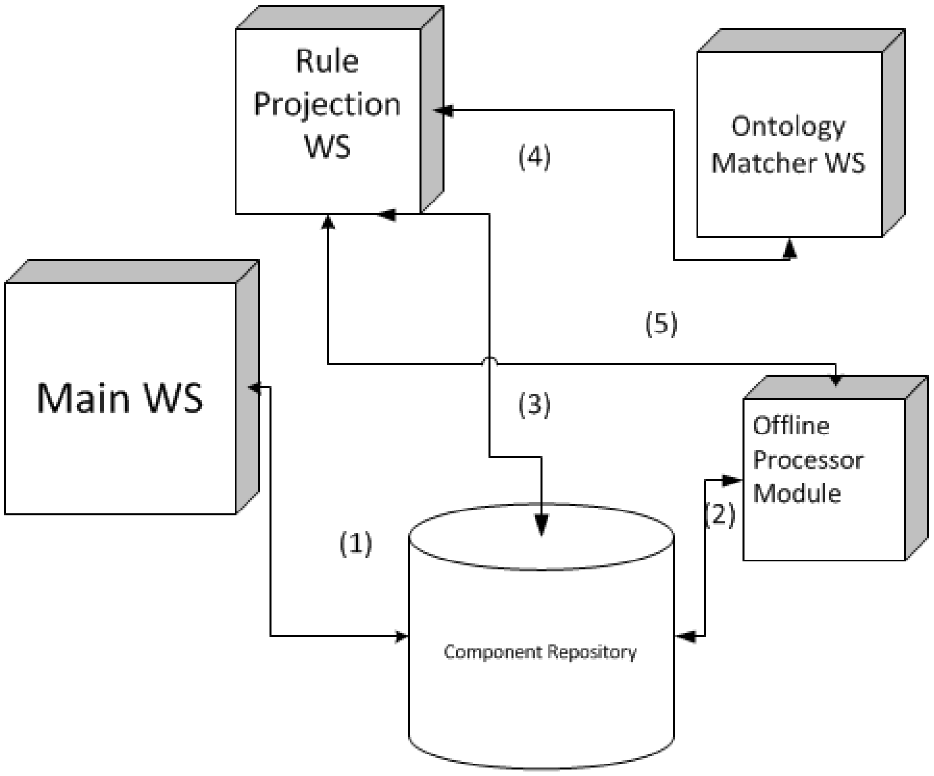


Figure 7 SODA- General view of the architecture

Whenever the ontology URI submitted by the client application as part of the request is available on the SODA repository, the output is returned immediately (or if a request for the same URI was previously processed by the system, the caching mechanism guarantees an immediate reply as well).
Otherwise, in the case of a newly introduced ontology URI, the system responds to the client by sending a ticket, which represents a unique identifier of the request (UID). By using this ticket the client can resubmit a query to the server from time to time to check if its request has been satisfied or not. In this case, when having a newly registered ontology, the system immediately passes on the request to a dedicated semantic coordination service in charge of processing the request and of interacting with other specialized components to accomplish the task.
The interface of this web service is presented in Figure 8; here we provide a detailed description of all of its methods:

registerClientInformation

The *registerClientInformation* method has the following elements:

INPUT:

- *requestType* – indicates weather the client is a ***lazy*** client or a ***developer***. We will discuss more in the next section the importance of such a parameter.
- *operatingSystem* – indicates the operating system of the client platform. This parameter is important because some components that are stored in the repository might have restrictions regarding some Operating Systems.
- *ontologyURI* – the URI of the client's *target ontology*.

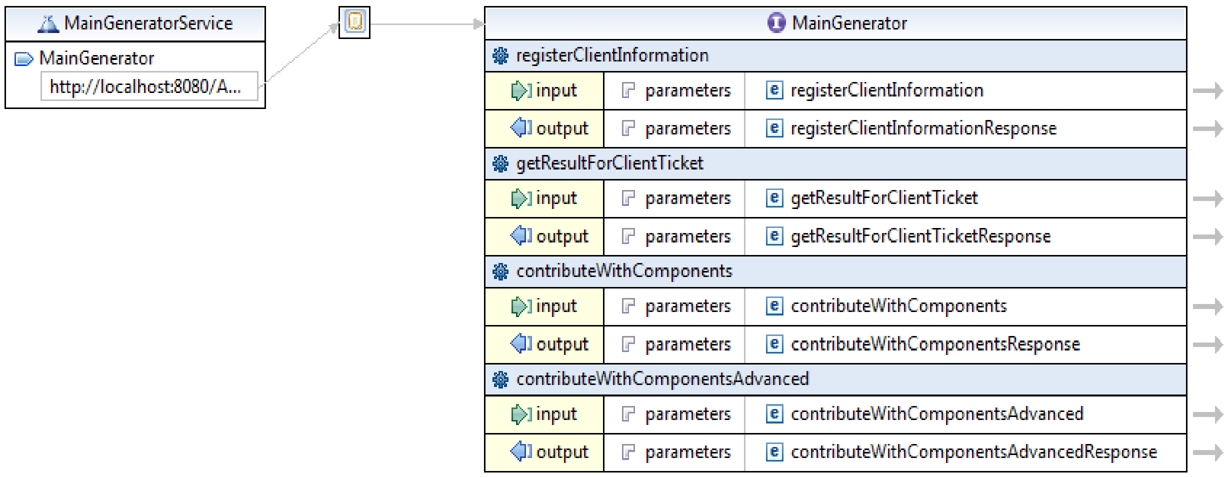

MainGeneratorService
MainGenerator
http://localhost:8080/A...
MainGenerator
registerClientInformation
input
parameters
registerClientInformation
output
parameters
registerClientInformationResponse
getResultForClientTicket
input
parameters
getResultForClientTicket
output
parameters
getResultForClientTicketResponse
contributeWithComponents
input
parameters
contributeWithComponents
output
parameters
contributeWithComponentsResponse
contributeWithComponentsAdvanced
input
parameters
contributeWithComponentsAdvanced
output
parameters
contributeWithComponentsAdvancedResponse

Figure 8 SODA- Main Web Service Module Interface

OUTPUT:

- *URIList* – represents the list of URIs of the individual UIMA Pear components. The list is returned as a comma separated string.
- *aggregatedDescriptor* – returns the content of the aggregated UIMA Analysis Engine (AAE). The AAE is obtained based on the individual components.
- *projectionRules* – represents the projection rules associated to the identify components, that project UIMA Features or TypeSystem into the classes of the client ontology.
- *prefixNamespaces* – the list of namspace prefixes, returned as a string.
- *clientTicket* – unique identifier for client request. If the ontology was not previously processed than the client can return in the system by using this UID to check if its request was completed.

ROLE:
This method records the information regarding the client request in the database. Afterwards, checks if the ontology URI was processed. In case it appears in the database, the associated components are retrieved and sent to the client under the format presented above. Otherwise, the request is recorded in the database with the status “Pending”. From time to time (10 minutes), the offline processor module checks the database and retrieves all the pending requests and processes them offline (through the Semantic Matching process cited above).

getResultForClientTicket

the *getResultForClientTicket* method is characterized by the following:

INPUT:

- *IdClientRequest* – represents the ticket received by the client when registering its request

OUTPUT:

- the same type of output as for the method presented above

ROLE:
The method queries the database and checks the status of the client request. If the status is “Processed” than the client receives the complete output (as presented in the method above), otherwise it is informed that the request has not yet been processed and is requested to retry after a while.

contributeWithComponent

The *contributeWithComponent* method enables users to contribute to the repository by uploading UIMA Pear components. Uploading a component does not require any projection document to be associated to it. The SODA repository is actually also a UIMA component repository and new UIMA components can be registered and stored in the SODA repository, independently from their use in SODA. Contributed UIMA components can later be associated to projection rules by other SODA contributors.

INPUT:

- *componentURI* – a string representing the URI of the component
- *componentDesc* – a string containing the description of the component.

OUTPUT:

- *feedback* – a string that informs the contributor whether the upload was performed successfully or not, or if the component is valid or not.

ROLE:
This method is targeted to extend the system’s data repository and to involve the UIMA community into the creation of a component repository. The contributor introduces into the system the URI of their UIMA Pear component and a short description of its functionalities. The system validates the component by trying to download and install it. If no errors occur and the component is valid the contributor is informed that their contribution was successfully recorded, otherwise they receive an error message and are asked to resubmit.

contributeWithComponentAdvanced

The *contributeWithComponentsAdvanced* method enables CODA clients to contribute to the repository by providing an ontology (referenced through its URI) and a set of projection rules and associated UIMA Component Pears for extracting and projecting data over it.

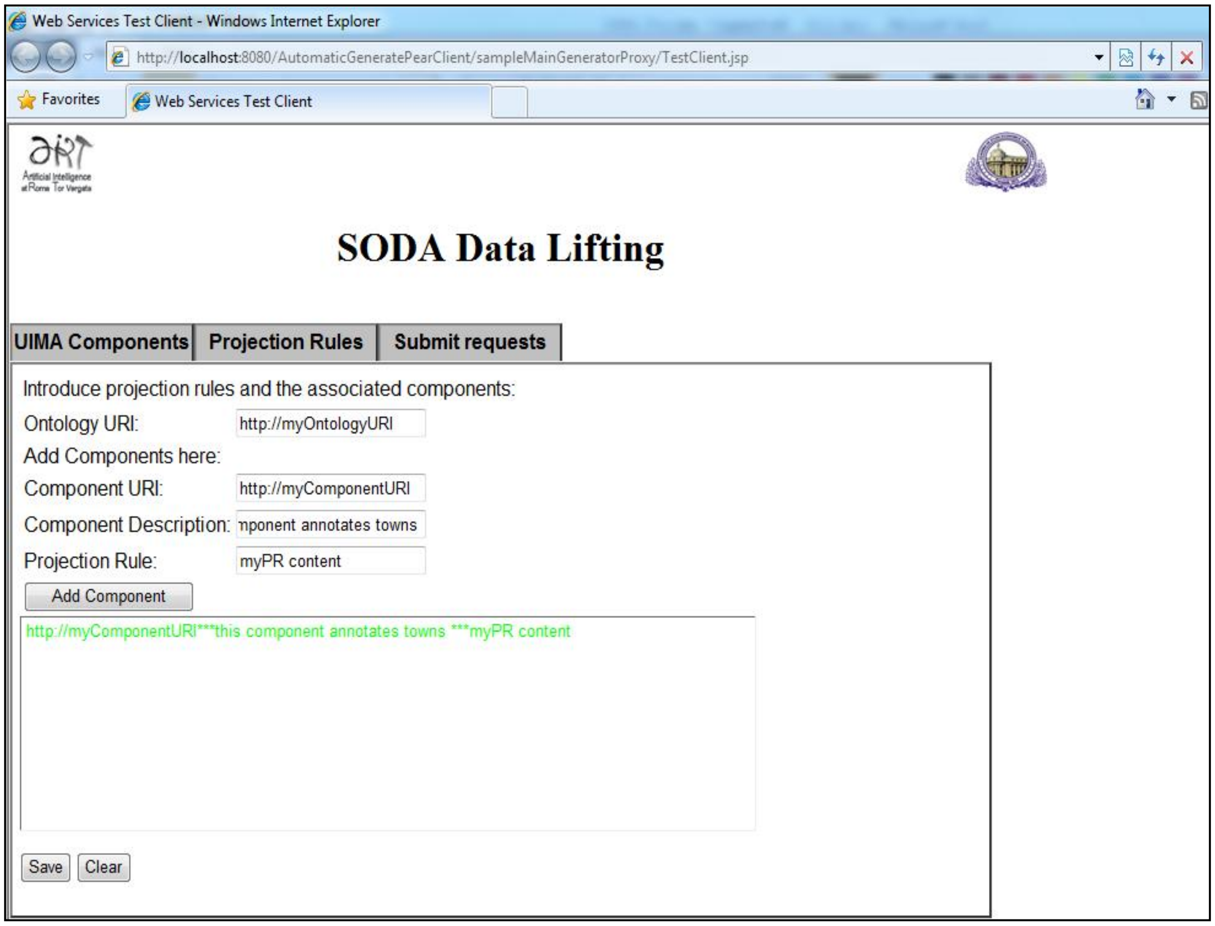
INPUT:

- *ontologyURI* – a string representing the URI of the client ontology
- *componentElementVector* – a string containing the serialized vector of UIMA Components URIs and their description and also the associated Projection rule.

OUTPUT:

- *feedback* – a string that informs the contributor whether the upload was performed successfully or not, or if the component is valid or not.

ROLE:
The purpose of this method is to allow a community of users to contribute the repository with new ontologies, extraction components and projection documents.
The system validates the components by trying to download and install them. If no errors occur and the components are valid then the contributor is informed that their contribution was successfully recorded, otherwise they receive an error message and are asked to resubmit.



Web Services Test Client - Windows Internet Explorer

http://localhost:8080/AutomaticGeneratePearClient/sampleMainGeneratorProxy/TestClient.jsp

SODA Data Lifting

Introduce projection rules and the associated components:

Ontology URI: http://myOntologyURI

Add Components here:

Component URI: http://myComponentURI

Component Description: nponent annotates towns

Projection Rule: myPR content

http://myComponentURI***this component annotates towns ***myPR content

Figure 9 A screenshot of the service page for contributing new components

Figure 9 presents a screenshot of the application dedicated to the components contributors.

Offline Processing Module

This module is in charge of processing clients' requests offline. It queries from time to time the database and retrieves all the pending requests. For each request, it first downloads the requested vocabulary (we assume the vocabulary is expressed in one of the W3C standards of the RDF family: RDF/RDFS/OWL/SKOS) and identifies all the concepts[xix] of the vocabulary.

Afterwards, based on the analyzed concepts from the ontology/thesaurus, the module tries to identify which UIMA components might be identified as possible matches based on the component description (in particular, on their *Type System*) and, if available, on the ontologies already associated to these components. For each identified component it retrieves from the database the ontology URI that was previously matched with it, in case there is one. For every ontology URI that was identified as possible match, the system performs a check. The check is done by submitting a request to an ontology matching module. After identifying the correct components, the system sends a request to the processor module.

This part is integrated in the projection rule services. The offline module actually calls the projection rule web service to obtain the projection rules by following a certain flow based on the client type (lazy or developer). The entire functionality of the projection rules web service is presented in detailed in a dedicated section.

The offline processor module is implemented as a job that can be scheduled. The current implementation uses the Quartz library.

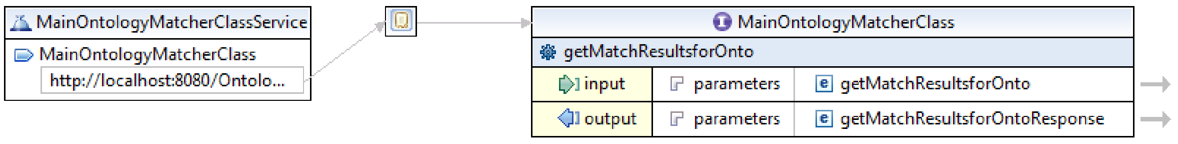


Figure 10 SODA Ontology matching web service interface

Ontology Matching Module

This module is in charge of identifying all the possible matching components between two ontologies which are identified based on their URL. For the time being we are only interested in identifying the classes that match.
Based on the results obtained from this module, the system performs an extra filtering of the set of possible UIMA Components that will serve for annotating the documents and also offer the input for the rule projection flow.
The interface of the web service is presented in figure 10.
The *getMatchResultsforOnto* method is characterized by the following elements:

INPUT:

- *requestOntoURI* – represents the ontology URI that the client uses
- *commaSeparatedPossibleOnto*- represents a comma separated string containing all the possible matching ontologies that are stored in the repository.

OUTPUT:

- responseMatch –a string that has the following format: ont1URI|_|(MatchClass11,origOntoClass11)!#(MatchClass12,origOntoClass12);ont2URI|_|(MatchClass21,origOntoClass21)!#(MatchClass22,origOntoClass22)

ROLE:
This method determines what are the matching components for each ontology that the client ontology is compared to.

Rule Projection Module

This module is implemented as a separate web service and it is used to create the projection rules.
The interface of the web service is presented in Figure 11
This web service has only one method, *projectRules*, that has as main purpose identifying the projection rules or creating new ones for UIMA components. The rules are developed in order to project UIMA features or TypeSystems into ontology classes for the beginning. The rules can afterwards be interpreted by CODA framework so that to enrich ontology. The entire flow will be presented in the next section.

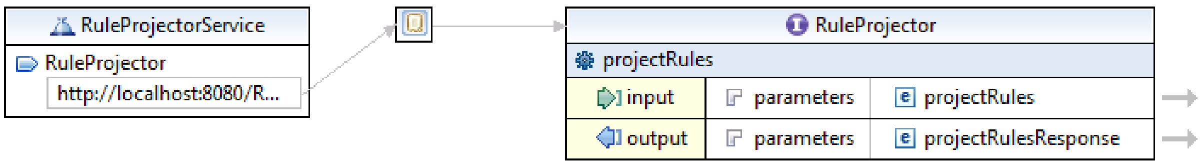


Figure 11 SODA-Rule projector web service

INPUT:

- *reqOntoURI* – the client ontology URI
- idClientRequest – represents the client ticket

OUTPUT:

- *projectionRule* – the projection rule created by the system according to the business rules that will be presented below.

ROLE:
When designing the projection rule the system first determines the client application type. The client application is classified into two categories: lazy or developer. If the client is a lazy one than the rules that will be sent to him will be only the ones that were previously validated, that is the ones that were already in the repository for the components identified as matching.
This method makes a call to the “Ontology Matching Web Service”. After retrieving the response, it parses it. For every component that was identified as a match, the system retrieves the rules from the database and changes the ontology class name within the rule. If the existing rule was a validated rule and the client was lazy or developer than the rule is saved in the database as associated to the client request. Otherwise, if the existing rule was not validated and the client was a lazy one, than the rule is not associated and the rule projection algorithm continues to investigate the other components.
For the components that were identified by the windows service as suitable for the ontology, but were not selected as matching components with other already existing ontology, the system creates new projection rules only if the client request was “developer”. The creation of the new rule consists in extracting the type systems, or the features that are present within the UIMA Analysis Engine from the UIMA Pear component. Afterwards, the features are projected into an ontology class. A simple sample rule can be seen below:

```
my=http://art.uniroma2.it/ontologies/st_example#;

rule it.uniroma2.art.uima.Author id:author1 0.9  {

        nodes = {
                person    uri    it.uniroma2.art.uima.Author:name
        }

        graph = {
                ?person   a    my:Person    .
        }

}
```

The rule above can be interpreted as follows: the entity annotated as *it.uniroma2.art.uima.Author* is projected into the target RDF, with a confidence of 0.9. The feature that is projected is *name*, the class to which it is projected is my:Person

FUTURE RESEARCH DIRECTIONS

The ambitions of such a complex framework as SODA is to embrace a wide community of users, establishing a milestone for industry-level Text Analysis and Content Management by fostering component re-use and fast deployment of information processing systems. In this sense, much of the future effort should be spent on actively supporting its adoption (e.g. by providing a bootstrapped repository of SODA projection rules and UIMA components accessible via SODA services; by documenting all the layers both at user and developer level to enable further contributions by communities of developers etc…).

However, the purpose of such an architecture, as a research effort, is also to lay the path for other approaches acting in the same direction, by riding state-of-the-art research and industry-level solutions to feed new ideas in it. In this sense, one of the foreseen actions for the future is to improve the semantic search functionalities based on semantic matching (currently based on naïve implementations of these techniques), by applying clustering and recommendation algorithms to the description of UIMA components, and to take into account the constraints imposed by the client systems, specified through the client request. The objective is to avoid the typical "go-off-from-your-working-environment-and-googlesearch your components" to enable instead fully-fledged semantic search services enabling components search and browsing of components which are semantically connected according to common purposes, data formats being used and other contradistinguishing features.

Another aspect which has to be improved is the relationship with other software provisioning frameworks, such as Maven: combining both efforts, probably reusing traditional provisioning frameworks for dependency resolution and physical provisioning while demanding to SODA search and component/rule harmonization can even better separate the specifications and have SODA benefit of already available solutions.

CONCLUSION

In this work we have introduced SODA, an architecture and a platform for supporting and automatizing the creation and population of semantic repositories and ontologies based on models of the RDF family. One characterizing aspect of SODA, which distinguishes it from other systems of this kind, is its real total openness: from component provisioning to extraction services, all of the layers which constitute SODA can be re-elaborated to suit specific needs and fit into different scenarios. This also assures data protection and privacy, where components contributed by the community may be downloaded and coordinated into private deployments of semantic data acquisition systems.

The objective of this contribution lies in the engineering of data acquisition processes, by offering formal languages and tools for driving information management from core raw sources to structured data, and by providing specifications and concrete frameworks for the realization of content processing and knowledge elicitation systems.

REFERENCES

Basili, R., Vindigni, M., & Zanzotto, F. (2003). Integrating Ontological and Linguistic Knowledge for Conceptual Information Extraction. *IEEE/WIC International Conference on Web Intelligence.* Washington, DC, USA.

Beckett, D., & Berners-Lee, T. (2008, January 14). *Turtle - Terse RDF Triple Language.* Retrieved from World Wide Web Consortium (W3C): http://www.w3.org/TeamSubmission/turtle/

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.

Buitelaar, P., Declerck, T., Frank, A., Racioppa, S., Kiesel, M., Sintek, M., et al. (2006). LingInfo: Design and Applications of a Model for the Integration of Linguistic Information in Ontologies. *OntoLex06.* Genoa, Italy.

Buitelaar, P., Olejnik, D., & Sintek, M. (2004). A Protégé Plug-In for Ontology Extraction from Text Based on Linguistic Analysis. *Proceedings of the 1st European Semantic Web Symposium (ESWS).* Heraklion, Greece.

Buyko, E., Chiarcos, C., & Pareja Lora, A. (2008). Ontology-Based Interface Specifications for an NLP Pipeline Architecture. *In: LREC 2008 - Proceedings of the 6th International Conference on Language Resources and Evaluation.* Marrakech, Morocco.

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

Cerbah, F., & Daille, B. (2007). A Service Oriented Architecture for Adaptable Terminology Acquisition. In Z. Kedad, N. Lammari, E. Métais, F. Meziane, & Y. Rezgui (A cura di), *Natural Language Processing and Information Systems* (Vol. 4592, p. 420-426). Springer Berlin / Heidelberg.

Chaudhri, V. K., Farquhar, A., Fikes, R., Karp, P., & Rice, J. P. (1998). OKBC: A programmatic foundation for knowledge base interoperability. *In Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)* (pp. 600-607). Madison, Wisconsin, USA: MIT Press.

Cimiano, P., & Völker, J. (2005). Text2Onto - A Framework for Ontology Learning and Data-driven Change Discovery. *Proceedings of the 10th International Conference on Applications of Natural Language to Information Systems*, (pp. 227-238). Alicante.

Cimiano, P., Haase, P., Herold, M., Mantel, M., & Buitelaar, P. (2007). LexOnto: A Model for Ontology Lexicons for Ontology-based NLP. *In Proceedings of the OntoLex07 Workshop (held in conjunction with ISWC'07).*

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

Cunningham, H., Maynard, D., Bontcheva, K., & Tablan, V. (2002). GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. *In Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02).* Philadelphia.

Dodinet, G. (2005, May 24). *Exploiting Maven in Eclipse.* Retrieved from IBM developerWorks.

Ferrucci, D. (2009, March 2). Unstructured Information Management Architecture (UIMA) Version 1.0. (A. Lally, K. Verspoor, & E. Nyberg, Eds.) OASIS Standard.

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.

Gennari, J., Musen, M., Fergerson, R., Grosso, W., Crubézy, M., Eriksson, H., et al. (2003). The evolution of Protégé-2000: An environment for knowledge-based systems development,. *International Journal of Human-Computer Studies, 58*(1), 89–123.

Griesi, D., Pazienza, M., & Stellato, A. (2007). Semantic Turkey - a Semantic Bookmarking tool (System Description). In E. Franconi, M. Kifer, & W. May (A cura di), *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, Proceedings. Lecture Notes in Computer Science. 4519*, p. 779-788. Springer.

Harman, D. (1992). The DARPA TIPSTER project. *SIGIR Forum, 26*(2), 26-28.

Pazienza, M. T., Stellato, A., & Turbati, A. (2010). A Suite of Semantic Web Tools Supporting Development of Multilingual Ontologies. In G. Armano, M. de Gemmis, G. Semeraro, & E. Vargiu (Eds.), *Intelligent Information Access. Studies in Computational Intelligence Series.* Springer-Verlag.

Pazienza, M., Scarpato, N., Stellato, A., & Turbati, A. (2008). Din din! The (Semantic) Turkey is served! *Semantic Web Applications and Perspectives.* Rome, Italy.

Pazienza, M., Stellato, A., & Turbati, A. (2008). Linguistic Watermark 3.0: an RDF framework and a software library for bridging language and ontologies in the Semantic Web. *Semantic Web Applications and Perspectives, 5th Italian Semantic Web Workshop (SWAP2008).* FAO-UN, Rome, Italy.

Peter, H., Sack, H., & Beckstein, C. (2006). SMARTINDEXER – Amalgamating Ontologies and Lexical Resources for Document Indexing. *Workshop on Interfacing Ontologies and Lexical Resources for Semantic Web Technologies (OntoLex2006).* Genoa, Italy.

Peters, W., Montiel-Ponsoda, E., Aguado de Cea, G., & Gómez-Pérez, A. (2007). Localizing Ontologies in OWL. *In Proceedings of the OntoLex07 Workshop (held in conjunction with ISWC'07).*

Velardi, P., Navigli, R., Cucchiarelli, A., & Neri, F. (2005). Evaluation of ontolearn, a methodology for automatic population of domain ontologie. In *Ontology Learning from Text: Methods, Applications and Evaluation.* IOS Press.

W3C. (2004). *OWL Web Ontology Language*. Tratto da http://www.w3.org/TR/owl-features/

W3C. (2009, August 18). *SKOS Simple Knowledge Organization System Reference.* Retrieved from World Wide Web Consortium (W3C).

Wilkinson, M., Vandervalk, B., & McCarthy, L. (2009). SADI Semantic Web Services - ‘cause you can’t always GET what you want! *in proceedings of the IEEE International Workshop on Semantic Web Services in Practice*, (pp. pp 13-18). Singapore.

Witte, R., & Gitzinger, T. (2009). Semantic Assistants -- User-Centric Natural Language Processing Services for Desktop Clients. *3rd Asian Semantic Web Conference (ASWC 2008). 5367*, pp. 360-374. Bangkok, Thailand: Springer.

Witte, R., & Papadakis, N. (2009). Semantic Assistants: SOA for Text Mining,. *CASCON 2009 Technical Showcase.* Markham, Ontario, Canada.

KEY TERMS & DEFINITIONS

Analysis Engine: a program that analyzes different kinds of documents (text, video, audio, etc) and infers[xx] information about them, and which implements the UIMA Analysis Engine interface Specification

CAS the UIMA Common Analysis Structure is the primary data structure which UIMA analysis components use to represent and share analysis results. It contains:

- The artifact. This is the object being analyzed such as a text document or audio or video stream. The CAS projects one or more views of the artifact. Each view is referred to as a *Sofa* (*Subject OF Analysis*).
- A type system description – indicating the types, subtypes, and their features
- Analysis metadata – “standoff” annotations describing the artifact or a region of the artifact
- An index repository to support efficient access to and iteration over the results of analysis.

CAS Consumer A component that receives each CAS in the collection, usually after it has been processed by an Analysis Engine. It is responsible for taking the results from the CAS and using them for some purpose, perhaps storing selected results into a database, for instance. The CAS Consumer may also perform collection-level analysis, saving these results in an application-specific, aggregate data structure.

UIMA pear: an archive file that packages up a UIMA component its code, descriptor files and other resources required to install and run it in another environment

web service: a software system designed to support interoperable machine-to-machine interaction over a network which exposes a standard interface in WSDL.

annotator components repository: environment that assures safe storage of different annotator components and also enables the retrieval and editing processes.
platform independence: a model of a software system , that is independent of the specific technological platform used to implement it.

---

[i] http://www.eqentia.com/
[ii] http://www.evri.com/
[iii] http://www.opencalais.com/
[iv] http://www.zemanta.com/
[v] OpenCalais privacy policy reported on http://www.opencalais.com/privacy fully respects the privacy of users with respect to their submitted documents, though strictly-secured documents are clearly not allowed to pass through a web service like that. This is just one of the many cases in which strict privacy policies on the user side end up in the demand for in-house solutions because those offered by free and open services on the web are not able to meet high-privacy requirements
[vi] http://uima.apache.org
[vii] http://nlp.stanford.edu/software
[viii] http://incubator.apache.org/opennlp/
[ix] http://uima.apache.org/
[x] http://ant.apache.org/ivy/
[xi] http://maven.apache.org/
[xii] http://www.eclipse.org/orbit/
[xiii] http://ant.apache.org/
[xiv] The term ontology here is used in a wide sense, covering every data structures from simple vocabularies/thesauri to highly structured content such as OWL ontologies
[xv] See graph pattern specification at: http://www.w3.org/TR/rdf-sparql-query/#GraphPattern
[xvi] See http://www.w3.org/TR/rdf-sparql-query/#construct for more details
[xvii] https://addons.mozilla.org/it/firefox/addon/8880 is the official page on Firefox add-ons site addressing Semantic Turkey extension, while http://semanticturkey.uniroma2.it/ provides an inside view about Semantic Turkey project, with updated downloads, user manuals, developers support and access to ST extensions
[xviii] Obviously, this does not prevent from the upload (and consequent download from users) of malicious components sending information to a remote server. On the other hand, the user is free to disconnect the processing machine from the web (or block SODA through a firewall) and thus prevent any undesired flow of information
[xix] In this context, with the term concept we refer to RDFS/OWL *classes* and SKOS *concepts*
[xx] We preferred to leave the definition from the UIMA Glossary, from the Overview and Setup guide available at: http://uima.apache.org/documentation, though we disagree with the use of the term "infers", as such terminology may evoke processes of formal inference, which are mostly distant from the kind of approaches followed in building Analysis Engines