IFAC-TSRR vol. 2 (2010) 159-167

On a Java based implementation of ontology evolution processes based on Natural Language Processing

Francesco Gabbanini([1]),

([1]) IFAC-CNR, Via Madonna del Piano 10, 50019 Sesto Fiorentino (FI), Italy

ICT.P10.007.001 – Responsabile Scientifico della Ricerca: Laura Burzagli

1 - Introduction

An architecture was described Burzagli et al. (2010) that can serve as a basis for the design of a *Collective Knowledge Management System*. The system can be used to exploit the strengths of collective intelligence and merge the gap that exists among two expressions of web intelligence, i.e., the Semantic Web and Web 2.0. In the architecture, a key component is represented by the *Ontology Evolution Manager*, made up with an *Annotation Engine* and a *Feed Adapter*, which is able to interpret textual contributions that represent human intelligence (such as posts on social networking tools), using automatic learning techniques, and to insert knowledge contained therein in a structure described by an ontology.

This opens up interesting scenarios for the collective knowledge management system, which could be used to provide up to date information that describes a given domain of interest, to automatically augment it, thus coping with information evolution and to make information available for browsing and searching by an ontology driven engine.

This report describes a Java based implementation of the Ontology Evolution Manager within the above outlined architecture.

2 - Specification of building blocks

The Ontology Evolution Manager (sketched in Fig. 1) is designed to take corpora of textual documents as input, produce a series of RDF statements and use them to enrich an ontology. In order to achieve these aims, main issues that have to be faced for its implementation consist in:

1. extracting machine readable knowledge from text;
2. transform extracted information in a form that is suitable for insertion into an ontology.

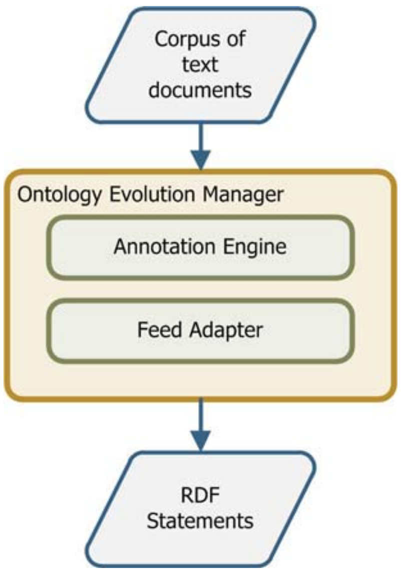

Fig. 1 - Ontology Evolution Manager general scheme

It is to be noted that knowledge to be extracted may in principle regard both assertions about individuals (as in “hotel X has a good cuisine”, where “hotel X” represents an individual of the “hotels” entity) and assertions about entities (as in “hotels have bedrooms”). In the first case the problem under study is named “ontology population”, while in the second it is named “ontology learning”, being the two terms synthesized by the more general “ontology evolution”.

In the following sections a description will be given about a Java based implementation of software components used for information extraction from texts (both in the sense of “ontology population” and “ontology learning”, see the Annotation Engine block in Fig.1) and for growing the knowledge base (see the Feed Adapter block in Fig. 1).

3 - Annotation Engine: implementation of Natural Language Processing techniques

In order to extract knowledge from texts, Natural Language Processing (NLP) techniques have to be employed (see Buitelaar and Cimiano, 2008).

Generally, these consist in a series of steps in which text is analysed, and include annotating text with a variety of information. These steps are typically represented by:

- splitting sentences;
- splitting text into words through tokenization;
- part-of-speech tagging (POS), which consists in a form of grammatical tagging, marking up the words in a text as corresponding to a particular part of speech, based on both its definition, as well as its relationship with adjacent and related words in a phrase, sentence, or paragraph;
- term indexing using gazetteers (particular sort of dictionaries);
- user-defined transduction processes to further analyse texts using finite state transducers that operate over annotations based on regular expressions, for pattern-matching, semantic extraction, and other operations over syntactic trees produced by the previous steps.

The Annotation Engine block is meant to implement NLP techniques so as to process and annotate textual contents, in order to provide coherent and structured inputs to the Feed Adapter block, which in turn uses them to enrich an ontology with new concepts and assertions.

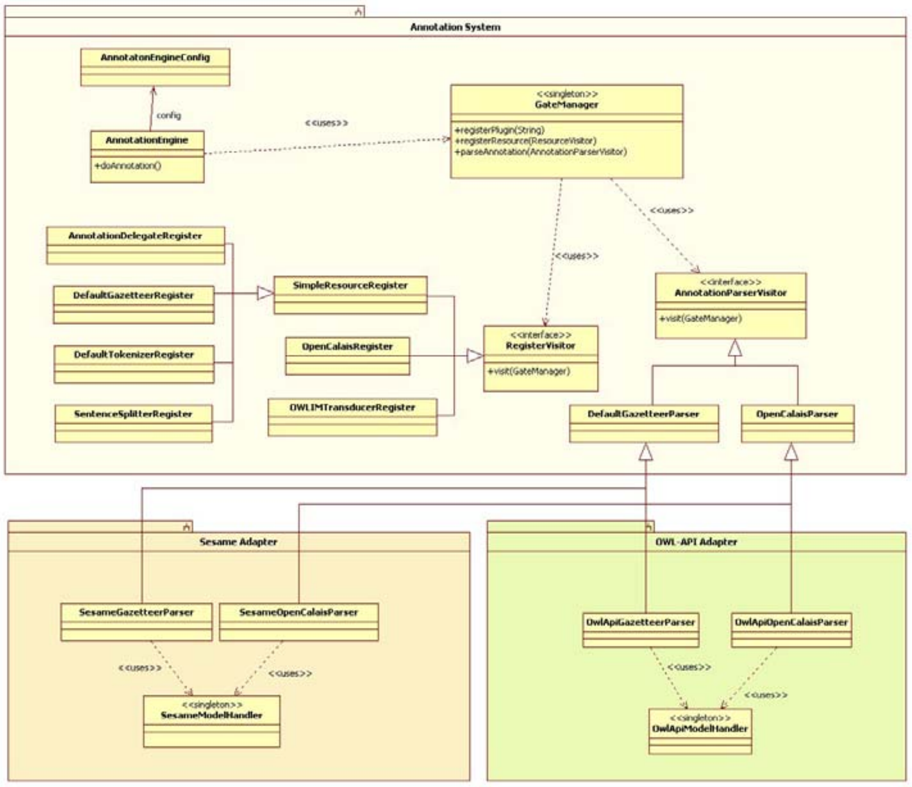

Fig. 2 - UML class diagram of the Ontology Evolution Manager

3.1 - *A GATE based implementation of NLP*

Although several algorithm implementations exist that perform some of the processing steps described in the previous section (see LExO (2010), Ontomat (2010), OpenNLP (2010), Text2Onto (2010)), the Annotation Engine block is based on the General Architecture for Text Engineering (GATE, see Cunningham et al. (2002), Maynard et al. (2008)).

GATE provides a modular object-oriented framework implemented in Java to embed language processing functionality in diverse applications. It can be extended and customised for different tasks by loading plugins, which can in turn contain a number of resources able to hold linguistic data and to process data. GATE is distributed with an Information Extraction (IE) system called “A Nearly-New IE System” (ANNIE), which relies on finite state algorithms and the Java Annotation Patterns Engine (JAPE) to process text corpora and performs operations such as sentence detection, tokenization, POS-tagging, chunking and parsing, named-entity detection, and pronominal co-reference.

JAPE is a finite state transducer system that operates over annotations based on regular expressions. Thus it is useful for pattern-matching, semantic extraction and many other operations over syntactic trees such as those produced by natural language parsers. JAPE operations are described by grammars which get converted into finite state machines as soon as they are loaded into the GATE framework.

Functionalities offered by the GATE APIs were reorganized in order for them to be available and easily usable within the more general framework of the Collective Knowledge Management System.

For this purpose, the `GateManager` Java class, as sketched in the UML diagram of Fig. 2, was designed to act as a *façade* to access various functionalities made available by the GATE API, such as initializing the GATE system, registering GATE plugins and resources, managing text corpora, parsing text corpora.

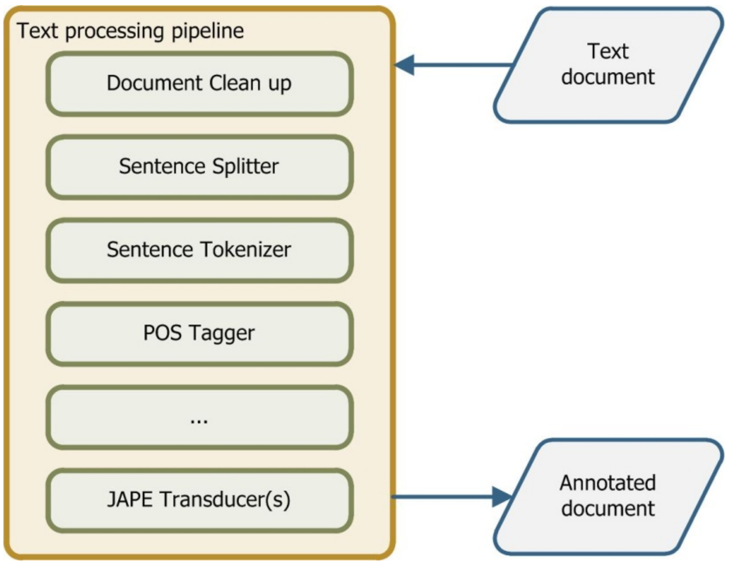

Fig. 3 - An example text processing pipeline

The `GateManager` class is responsible of managing the annotation process, which is based on a pipeline approach. A text document enters the pipeline and gets processed by the various registered plug-in resources, which, in turn, enrich it with a set of annotations. Each plug-in may contain a set of text annotation resources which can take advantage of annotations taken by means of resources that precede it in the pipeline. An example pipeline is given in Fig. 3.

As the text document enters the pipeline it first gets cleaned up from previous annotations; then sentences (through a Sentence Splitter resource) and words (through a Sentence Tokenizer resource) are identified; then POS tagging is performed. These constitute fundamental steps on which further processing steps can be based and which may include applying gazetteers to recognize geographic entities, proper nouns or dates. Finally, user defined elaboration processes are performed at the end of the pipeline, by means of transducers that use custom JAPE grammars that allow identifying patterns that are relevant for a certain domain and annotating them. At the end of the text processing pipeline an annotated text document is obtained and the GATE API includes Java classes that allow going through the annotations.

With reference to the UML diagram in Fig. 2, showing the underlying architecture of the natural language processing infrastructure, the `GateManager` class is responsible of managing the annotation engine: for this purpose it needs plug-ins and resources to be registered for pipeline annotation, each resource implementing an annotation step (see blocks within the text processing pipeline in Fig. 3). This process is implemented using a *visitor* pattern (see Gamma et al., (1995)).

The `GateManager` is first made aware of which plug-ins to use and of which resources (taken from previously set plug-ins) to use for the set-up of the pipeline. Each resource is modelled by a register class which implements a `RegisterVisitor` interface and is capable of performing self-initialization steps. Register classes may require or not property maps for initialization purposes: in the latter case they consist in a generalization of the `SimpleResourceRegister` class. More complex annotation processes require custom register classes to be written. As an example, annotations based on custom JAPE grammars are performed using objects of class `OWLIMTransducerRegister`. The class can be initialized by specifying custom JAPE grammars to be used to identify patterns that are relevant for a certain domain and to annotate texts based on the occurrence of these patterns. Obviously, multiple instances of `OWLIMTransducerRegister` may be inserted into the pipeline.

Through the `visit` method of its interface, each register class is added to a `SerialAnalyserController` object, which is defined by a Java class in the GATE API and is used to manage the text processing pipeline (see code excerpt in Tab. 1).

Tab. 1 - Code excerpt showing how to implement a text processing pipeline

```
      public class GateManagerTest {
        protected GateManager gateManager;
          ...
          public void initializationTest() {
          gateManager = GateManager.getInstance();
          ...
          gateManager.registerPlugin("ANNIE");
          gateManager.registerPlugin("Tagger_OpenCalais");
          ...
          gateManager.registerResource(new
    AnnotationDeleteRegister());
          gateManager.registerResource(new
    SentenceSplitterRegister());
          gateManager.registerResource(new
    DefaultTokenizerRegister());
          gateManager.registerResource(new
    DefaultGazetteerRegister());
          gateManager.registerResource(new
    OpenCalaisRegister("..."));
          ...
          Corpus elaboratedCorpus =gateManager.elaborateCorpus();

          DefaultGazetteerParser gazetteerParser = new
            DefaultGazetteerParser();
          gateManager.parseAnnotation(gazetteerParser);
          gazetteerParser.getAnnotatedResources();
          ...
        }
      }

      public class GateManager {
        private static GateManager instance;
        private SerialAnalyserController serialController;
        public   void   registerResource(RegisterVisitor   register)
  throws GateException {
          register.visit(this);
        }
        ...
        public  void  parseAnnotation(AnnotationParserVisitor  parser)
  throws GateException {
          parser.visit(this);
        }
      }

      public      class      DefaultGazetteerRegister      implements
 RegisterVisitor {
      private String resource = "...";
        public  void  visit(GateManager  manager)  throws  GateException
  {
          new SimpleResourceRegister(resource).visit(manager);
        }
        ...
      }

      public      class      DefaultGazetteerParser      implements
 AnnotationParserVisitor {
        public  void  visit(GateManager  manager)  throws  GateException
  {
```

```
            //annotation parsing code
            ...
          }
          public List<AnnotatedResource> getAnnotatedResources() {
            return resources;
          }
        }
```

After initialization, the `GateManager` is ready to perform text processing by running all the registered resources in cascade. Once text processing is made, the system ends up with a corpus of annotated documents: these can be parsed using an effective class infrastructure which was setup, again, using the visitor pattern. A parser interface was created, named `AnnotationParserVisitor`, to be implemented by annotation parser classes that have to be set up for each type of annotation. Annotations are retrieved from the documents by issuing a call to the `parseAnnotation` method of the `GateManager`, which takes a parser object as input.

This class structure efficiently encapsulates various GATE functionalities and allows to conveniently separate plugin and resources initialization from their usage in the pipeline, and to conveniently retrieve annotations from the corpus as object of class `AnnotatedResource`.

Finally, it is to be noted that the whole annotation process can be also managed from a single entry point, i.e., the `AnnotationEngine` class, which can be configured using an `AnnotationEngineConfig` object and holds a static reference to the `GateManager`.

Annotations made available from the different resources constitute the basis on which ontology evolution is performed by the Feed Adapter, as they in principle contain new concepts, relations or individuals relevant to the domain under study.

4 - Feed Adapter: the ontology evolution block

A number of Java based frameworks exist to create, alter and persist ontologies. As each one has different characteristics, they suit best for different application scenarios.

Before designing the Feed Adapter block, the most popular semantic web frameworks were examined. Characteristics of interest that were considered are reported in Tab. 2.

Tab. 2 - Characteristics of semantic web frameworks

Name	**SPARQL support**	**OWL 2.0 support**	**Reasoning features**	**Persistence**
Jena 2.6.2	Yes	No	Unable to reason on data type restrictions (the API is not compatible with a version of Pellet that is capable of reasoning on data type restrictions)	file, database
Protegé OWL API	No	No	Unable to reason on data type restrictions (the API is not compatible with a version of Pellet that is capable of reasoning on data type restrictions)	file
OWL API 3.0.0	No	Yes	Able to reason on data type restriction, if the HermiT reasoned is used, as Pellet is still not compatible with OWL API 3	file
AllegroGraph 3.3	Yes	?	Able to reason on data type restriction. Uses a proprietary reasoner (RDFS++), which is a RDF reasoner and not an OWL reasoner	database
Sesame 2.3.1	Yes	Yes	Unable to reason on data type restrictions. OWLIM is compatible with Sesame, but it is only an OWL Lite reasoned	file, database (MySql, Postgres),

	binary files

Desirable features for the Feed Adapter implementations are represented by:

- Support for OWL 2, which represents the most recent recommendation (dated 27th October 2009) of W3C that refines and extends OWL, the Ontology Web Language, see OWL Working Group at W3C (2009);
- ability to reason and make inference over data type restrictions;
- support for a variety of persistence methods;
- support for SPARQL Protocol and RDF Query Language (SPARQL, see SPARQL Working Group at W3C (2008)) queries.

Unfortunately, as the table shows, among the most popular products in this sector, no "full featured" framework is available.

However, the most "promising" frameworks to be adopted within the Collective Knowledge Management System were identified to be OWL API 3 and Sesame 2.3.1 1 (see OWL API (2010) and Sesame (2010), respectively, both of them Open Source), also considering the fact that they are supported by an active community of developers.

In order not to be tied to a particular implementation and to a precise framework, the Feed Adapter was designed as a middleware block acting as an adapter between annotations, coming from parsers described in section 0, and an ontology. For the moment only the adapter for the Sesame 2.3.1 framework was implemented, with the OWL API 3 implementation being in progress.

4.1 - *Sesame Adapter implementation details*

The Sesame Adapter is designed around the `SesameModelHandler` and includes the `SesameGazetteerParser` and `SesameOpenCalaisParser` classes.

The `SesameModelHandler` maintains a reference to a `Repository` interface, which is part of the Sesame API and can be used to access various `Repository` implementations, such as the `SailRepository` (also part of the Sesame API), which defines a Sesame repository that contains RDF data that can be queried and updated and operates on a stack of `Sail` objects. `Sail` objects can store RDF statements and evaluate queries over them.

Through the `SesameModelHandler` it is therefore possible to get access to statements that are present in the repository and to modify the repository itself.

As for the `SesameGazetteerParser` and `SesameOpenCalaisParser`, these are extensions, respectively, of the `DefaultGazetteerParser` and `OpenCalaisParser`, of which they override the `visit` method: this allows mapping annotations coming from the NLP process to assertions in the ontology. Although implemented only for the previously mentioned parsers, this construct may be generalised to any kind of parser.

A usage sample of the adapter is given in Tab. 3, which illustrates an excerpt from a JUnit test case which also represents an example of how to use the framework for an ontology evolution process. It is to be noted that the natural language processing step is centrally managed through the `AnnotationEngine` class, whereas in Tab. 1 it was handled through the `GateManager`.

Tab. 3. Code excerpt showing how to implement the ontology evolution process

```
      public class OntoEvolutionTest {
        private AnnotationEngine annotationEngine;
          private        final        String        BASE URI        =
  "http://www.ifac.cnr.it/test#";
        private final String REPOSITORY_PATH = "nativeStore/owlim";

        @Before
        public void setUp() throws Exception {
          GateManager.getInstance();

    SesameModelHandler.getInstance().createOWLIMRepository(REPOSITORY PA
TH);

         AnnotationEngineConfig          config          =          new
AnnotationEngineConfig.Builder()
                       .pluginName("ANNIE")
                       .pluginName("Tagger_OpenCalais")
                       .resourceRegister(new AnnotationDeleteRegister())
                       .resourceRegister(new SentenceSplitterRegister())
                       .resourceRegister(new DefaultTokenizerRegister())
                       .resourceRegister(new POSTaggerRegister())
                       .resourceRegister(new DefaultGazetteerRegister())
                       .resourceRegister(new OpenCalaisRegister(...))
                       .resourceRegister(new OWLIMTransducerRegister())
                       .annotationParser(new
SesameGazetteerParser(BASE_URI))
                       .annotationParser(new
SesameOpenCalaisParser(BASE_URI))
                       .textToAnnotate(text)
                       .build();
         this.annotationEngine = new AnnotationEngine(config);
       }

       @Test
       public void testDoAnnotation() throws Exception {
         this.annotationEngine.doAnnotation();

         RepositoryConnection               connection               =
SesameModelHandler.getInstance().getRepository().getConnection();
         ValueFactory                 valueFactory                   =
SesameModelHandler.getInstance().getRepository().getValueFactory();
         try {
           URI aURI = valueFactory.createURI(BASE_URI, "...");
           RepositoryResult<Statement>           statements          =
connection.getStatements(aURI, OWL.INDIVIDUAL, null, true);
           ...
         } finally {
           connection.close();
         }
       }
     }
```

5 - Conclusions and future developments

The report illustrates details regarding the Java implementation of an Ontology Evolution Manager, which is a software that extracts structured information from natural language and uses it for “growing” ontologies.

As such, it aims at exploiting synergies between Web 2.0 and the Semantic Web, potentially acting as a bridge from user contributed (unstructured) text to information organized in ontologies.

Future work implementation work related to the processing logic layer of the knowledge management system will regard enriching the framework with support for relations discovery using WordNet (see WordNet (2010)) and the Scarlet (see Scarlet (2010)) framework. As for ontology management, it will be interesting to evaluate Empire (see Empire (2010)), which is an implementation of the Java Persistence API (JPA) for RDF and the Semantic Web. Adoption of JPA for persistence would represent a step ahead towards integration of the collective knowledge management framework within Java Enterprise Edition applications.

References

1. Berners-Lee, T., Hendler, J., Lassila, O., 2001. The Semantic Web. Scientific American 284 (5), 34-43.
2. Buitelaar, P., Cimiano, P. (Eds.), 2008. Ontology Learning and Population: Bridging the Gap between Text and Knowledge. Vol. 167 of Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam.

3. Burzagli, L., Como, A., Gabbanini, F., 2010. Towards the convergence of Web 2.0 and Semantic Web for e-Inclusion. In: Miesenberger, K., Klaus, J., Zagler, W., Karshmer, A. (Eds.), Computers Helping People with Special Needs. Vol. 6180 of Lecture Notes in Computer Science. Springer, pp. 343-350.
4. 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.
5. Empire, 2010. Available at http://github.com/clarkparsia/Empire, last visited on 17/09/2010
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
7. LExO, 2010. Available at http://code.google.com/p/lexo/, last visited on 20/09/2010
8. Maynard, D., Li, Y., Peters, W., 2008. NLP techniques for term extraction and ontology population. In: Proceedings of the 2008 conference on Ontology Learning and Population: Bridging the Gap between Text and Knowledge. IOS Press, Amsterdam, The Netherlands, pp. 107-127.
9. Ontomat, 2010. Available at http://annotation.semanticweb.org/ontomat/index.html, last visited on 20/09/2010
10. OpenNLP, 2010. Available at http://opennlp.sourceforge.net/, last visited on 20/09/2010
11. OWL API, 2010. Available at http://owlapi.sourceforge.net/, last visited on 17/09/2010
12. OWL Working Group at W3C, 2009. http://www.w3.org/2007/OWL/wiki/OWL_Working_Group, last visited on 17/09/2010.
13. Scarlet, 2010. Available at http://scarlet.open.ac.uk/, last visited on 17/09/2010
14. Sesame, 2010. Available at http://www.openrdf.org/, last visited on 17/09/2010
15. SPARQL Working Group at W3C, 2008. SPARQL Query Language for RDF Recommendation. Available at http://www.w3.org/TR/rdf-sparql-query/, last visited on 17/09/2010
16. Text2Onto, 2010. Available at http://sourceforge.net/projects/texttoonto/, last visited on 20/09/2010
17. WordNet, 2010. Available at http://wordnet.princeton.edu/, last visited on 17/09/2010