Tackling interoperability issues within UIMA workflows

Nicolas Hernandez

LINA (CNRS - UMR 6241) – University of Nantes
2 rue de la Houssinière – B.P. 92208, 44322 NANTES Cedex 3, France
first.last@univ-nantes.fr

Abstract

One of the major issues dealing with any workflow management frameworks is the components interoperability. In this paper, we are concerned with the Apache UIMA framework. We address the problem by considering separately the development of new components and the integration of existing tools. For the former objective, we propose an API to generically handle TS objects by their name using reflexivity in order to make the components TS-independent. In the latter case, we distinguish the case of aggregating heterogeneous TS-dependent UIMA components from the case of integrating non UIMA-native third party tools. We propose a mapper component to aggregate TS-dependent UIMA components. And we propose a component to wrap command lines third party tools and a set of components to connect various markup languages with the UIMA data structure. Finally we present two situations where these solutions were effectively used: Training a POS tagger system from a treebank, and embedding an external POS tagger in a workflow. Our approch aims at providing quick development solutions.

Keywords: UIMA, interoperability, type system, data serialization format, software component integration

1. Introduction

Over the last few years, there has been growing interest in the *Apache Unstructured Information Management Architecture* [1] (UIMA) (Ferrucci and Lally, 2004) as a software solution to manage unstructured information. In comparison with *GATE*[2] (Cunningham, 2002), its probable major difference is that it was initiated more recently and not by researchers but by industrials (IBM) with stronger engineering and design skills. GATE presents the interests of being used for long in the Natural Language Processing (NLP) community and of offering a wide range of analysis components and tools. *NLTK*[3] remains also an interesting framework in particular because of the availability of numerous integrated third party tools and data resources, and because of its programming language –Python– which is well-adapted for handling text material and quick development.

From the NLP researcher point of view, Apache UIMA is an attractive solution for at least two reasons[4]: First, it dissociates the engineering middleware problems from the NLP issues and takes in charge many of the engineering needs like the workflow deployment, the data transmission or the data serialization; Second, it offers a programming framework for defining and managing NLP objects present in analysis tasks (such as creating or getting the annotations of a given type).

One of the major issues dealing with any workflow management frameworks is the components interoperability. UIMA components only exchange data. So the data structure of the shared data is important since it ensures the interoperability. UIMA offers mechanisms to freely define its own data structure and the means to handle it afterwards. This may lead to some interoperability problems since anyone can design its own domain model to represent the same concepts. For example, *word*, *mot* or *token* can be different names to mean the same type of information. In addition, as shown in Figure 1, an information, such as the part-of-speech (POS) value *Noun* of a word, can be represented in several ways. In (1), it is the value of a POS feature of a word annotation. In (2), it is the value of a whatever feature of a POS annotation at the same offset of a word annotation. And in (3), it corresponds itself to a type of an annotation covering the desired text offsets of a word. In the UIMA jargon, the definition of the data structure is called the type system (TS).

In this paper, we address the interoperability issue by considering separately the development of new components and the integration of existing software instruments. For the former objective, we propose an API to generically handle TS objects by their name using reflexivity in order to make the components TS-independent (Section 3.1.). In the latter case, we distinguish the case of aggregating heterogeneous TS-dependent UIMA components from the case of integrating non UIMA-native third party tools. We propose a mapper component to aggregate TS-dependent UIMA components (Section 3.2.). And we propose a component to wrap command lines third party tools (Section 3.3.) and a set of components to connect various markup languages with the UIMA data structure (Section 3.4.). Finally we present two situations where these solutions were effectively used: Training a POS tagger system from a treebank (Section 4.1.), and embedding an external POS tagger in a workflow (Section 4.2.).

Our approach aims at providing quick development solutions. This work is part of the efforts for building a French-speaking community around UIMA (Hernandez et al., 2010).

[1] `http://uima.apache.org`
[2] `http://gate.ac.uk`
[3] `http://www.nltk.org`
[4] See (Hernandez et al., 2010) for more reasons.

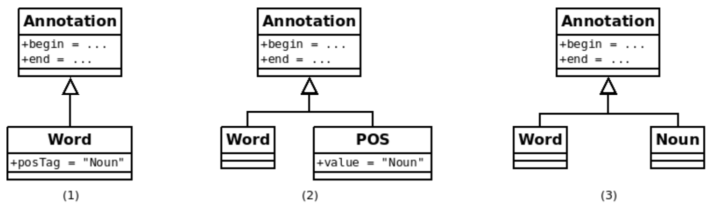

Figure 1: Various data model definitions.

2. Background

The question of interoperability for sharing language resources and technology is the concern of several related initiatives such as the CLARIN[5] project and the META-NET[6] network. Various kinds of related interoperability aspects can be considered: the «domain models» which provide definitions of the types of the data elements that constitute a domain as well as the description of how these types are structured in the domain, the «data serialization formats» which are used to store or to remotely transmit the primary data and its associated metadata, the «metamodels» which are abstractions of domain models. . .

2.1. UIMA concepts

In March 2009, the UIMA metamodel was voted as a standard by the OASIS consortium[7]. «The specification defines platform-independent data representations and interfaces for text and multi-modal components or services. The principal objective of the UIMA specification is to support interoperability among components or services.»
The *Common Analysis Structure* (CAS) is the data structure which is exchanged between the UIMA components. It includes the data, subject of analysis and called the *Artefact*, and the metadata, in general simply called the *Annotations*, which describe the data. The annotations are structured in *Views* (e.g. an HTML document can have the following views: the HTML structure, the extracted text, and a translation of the latter) where they are directly associated to. The annotations are made of a feature structure and are stored in CAS index. The definition of an annotation structure is called the *Type System* (TS) and consists of an implementation of a domain model.
A UIMA workflow is made of three types of components: the *Collection Reader* (CR) which imports the data to process (for example from the Web or from the file system...) and turns it into a CAS. The *Analysis Engines* (AE) which literally process the data (including but not restricted to NLP analysis tasks); The annotations result from AE processing. And lastly the *CAS Consumer* (CC) which exports the annotations (for example to a database or to an XML representation of the analysis results).
The UIMA framework handles the effective transmission of the data either as objects between components deployed on a same computer or as XML streams by (a)synchronous web services. In addition, the UIMA framework comes with components which offer the possibility to serialize the exchanged data into the XML Model Interchange[8] (XMI) format which is the OMG's XML standard for exchanging Unified Model Language (UML) metadata.

2.2. Serialization formats for exchanging data between third party tools

As part of the ISO's TC37 SC4, (Ide and Suderman, 2009) defend the idea that the GrAF (Graph Annotation Framework) format, which is the xml serialization of the LAF (Linguistic Annotation Framework) metamodel (Romary and Ide, 2004), can serve primarily as a «pivot» for transducing serialization formats. They show that it is possible to convert the information from the GrAF to a UIMA CAS. This is mainly made possible since both underlying metamodels are based on a graph structure. Nevertheless the conversion is not straightforward; Since the UIMA CAS defines typed feature structures, the process requires the use of external knowledge sources to be able to specify the types of the UIMA features. Neither the outcome is not completely bijective; The process requires the definition of UIMA additional features for being able to explicit the graph structure of the GrAF and consequently to reverse the process. As a consequence, any UIMA CAS cannot be transduced into a GrAF without an adaptation of its structure.
As a matter of fact the XMI format remains an appealing solution to store and exchange UIMA CAS.

2.3. Main trend for tackling the type system interoperability problem

The proposed solutions to tackle the domain model (i.e. type system) interoperability problems were similar to the solutions proposed for tackling the XML languages interoperability problems. The main trend was to define a common tool- and domain-free TS. In fact, several TS emerged: The CCP metamodel's TS (Verspoor et al., 2009), the DKPro's TS at the Darmstadt University[9] (Gurevych et al., 2007), the Julie lab's TS (Hahn et al., 2007)[10] and the U-Compare project's TS (Kano et al., 2009; Thompson et al., 2011)[11]. The former consists of a simple annotation hierarchy where the domain semantics is captured through pointers into external resources. The others roughly consist of an abstract hierarchy of NLP concepts covering the

[5]http://www.clarin.eu
[6]http://www.meta-net.eu
[7]www.oasis-open.org/committees/uima
[8]http://www.omg.org/spec/XMI
[9]http://www.ukp.tu-darmstadt.de/software/dkpro
[10]https://www.julielab.de/Resources/Software.html
[11]http://u-compare.org

various linguistic analysis levels.
In particular, (Thompson et al., 2011) defend the adoption of the Apache UIMA framework and of the integration U-Compare system within the META-SHARE infrastructure which is an initiative of META-NET for sharing language resources and technology on a range of European languages.
In practice these type systems are still in use separately.
As we mentioned in (Hernandez et al., 2010), as often as possible existing TS should be used, but in our opinion, distinct TS will always exist and it will always be necessary to develop software converters either to ensure compatibility with existing TS-dependent components or to fit with specific problem requirements. The U-Compare project, for example, comes with some ad hoc TS converters from CCP, OpenNLP and Apache which turn them into the U-Compare TS. It offers also some U-Compare TS to OpenNLP.

3. Handling more easily and generically the UIMA framework

Below we present the projects we develop for this purpose. They are made of libraries and UIMA components.

3.1. `uima-common`: Re-using common UIMA codes

`uima-common`[12] aims at assembling common and generic code snippets that can be usefully reused in several distinct UIMA developments (like AE or any applications). It is mainly made of two parts: A UIMA utilities library and a generic AE implementation.
The library defines methods to generically handle the various UIMA object types (i.e.view, annotation, feature) referring to them by string names. It centralizes redundant codes in particular for parsing collections of these objects, getting and setting them.
The generic AE can be used by extending its class. It allows to develop TS-independent AE and so to handle generically the processed views and annotations. This is made possible by specifying the names of the handled views and annotations by parameters. In addition, the AE follows a common analysis template: It allows to perform some process (e.g.. does it start with an upper case letter) on some annotations (e.g. the tokens)covered by some others (e.g. the sentences) which are only present in some views (e.g. the extracted text) by simply overwriting the right methods. The processing result can be stored in a new view, or a new annotation[13] only by setting their name by parameters.
In some way, it can be compared with *uimaFIT*[14] but `uima-common` is more centered on the internal development of components while uimaFIT is more dedicated to the development of applications with the ability to perform dynamic configuration and instantiation of components and workflows.
The implementation is hardly based on the `java.lang.reflect` API.

[12]`http://code.google.com/p/uima-common`
[13]It is also possible to set or update the feature value of an annotation.
[14]`http://code.google.com/p/uimafit`

3.2. `uima-mapper`: Mapping between UIMA objects

`uima-mapper`[15] aims at tackling two issues: mapping UIMA objects (annotations and features from distinct type systems and views) and recognizing annotation patterns. Both issues are indeed two views of a same process performed on the fly by declarative rules.
In NLP, the rule-based analysis is one of the two main approaches to process the documents; The other one is based on machine learning. The development of graph matching systems presents two difficulties: 1. designing simple and intuitive language for expressing rules and 2. implementing an effective engine to process the rules over graphs.
`uima-common` offers some solutions when developing new UIMA components to make them TS-independent. Nevertheless, many components available [16] assume the names of the view to work with and of the annotations to process or create. `uima-mapper` proposes to address this problem by inserting between two components using the same concepts defined with distinct TS, a specific AE which is able to *translate* annotations from the former TS to the latter TS.
The AE only requires to define the parameters of the rules file paths. A rule declares some edit operations (such as creation) by specifying constraints over an annotations pattern and over the annotations. On the creation operation it is possible to set features with values imported from the matched pattern. Currently the pattern can count only one annotation and only the creation operation is supported. As a matter of fact, this AE was first a proof of concept. The respect of the semantic coherence of the transformation is up to the user.
The implementation is based on *W3C XPath*[17] as the language to support the declaration of constraints over the source annotations and *Apache JXPath*[18] as the engine which processes the constraints.
JAPE of GATE (Thakker et al., 2009; Cunningham et al., 2011) and Unitex (Paumier, 2003) offer capabilities for defining regular expressions over annotations. Efforts have been performed to allow the embedding of GATE pipelines within UIMA[19]; and similarly with Unitex (Meunier et al., 2009)[20]. Even if the embedding of such tools can be a solution for mapping UIMA annotations, the approach seems too complex and resources consuming compared to the need. Indeed, it is necessary to both configure these tools as well as the embedding; In addition to having to write the mapping rules between UIMA objects, the user must also supply mapping rules defining how to map between UIMA and GATE/Unitex annotations.
Up to our knowledge, it exists currently two native UIMA projects whose goal is to offer capabilities for recognising UIMA annotations patterns in a UIMA workflow: The *zanzibar*[21] project and the *TextMarker* component (Kluegl

[15]`http://code.google.com/p/uima-mapper`
[16]`http://uima.apache.org/sandbox.html`
[17]`http://www.w3.org/TR/xpath`
[18]`http://commons.apache.org/jxpath`
[19]`http://gate.ac.uk/sale/tao/#x1-48400020`
[20]`http://sourceforge.net/p/gramlab`
[21]`http://code.google.com/p/zanzibar` and

et al., 2009)[22]. The former project has not been upgraded since March 2011. The current version is hardly stable. And despite the fact that patterns can be defined, the constraints over the annotation are quite limited in expressiveness (i.e. only the presence of a feature value can be specified). On the other hand, the TextMarker component is a very appealing project because of the rich expressiveness it aims at offering. It is now further developed and hosted at Apache UIMA and there is no current stable release and it is recommended to use the previous stable version which remains very complex to use and quite dependent of the Eclipse environment. In this previous version, we encountered the problem that TextMarker engine allowed only one annotation of the same type starting at the same offset. This limitation prevents us from using it since we could not combined it with the `uima-connectors`'s `XML2CAS` AE presented below (See Section 3.4.). The `XML2CAS` AE produces as many annotations of the same type as there are attributes in an XML element. There was no possibility to express TextMarker rules for selecting a specific annotation of this type; The TextMarker always considered the first one at this offset in the annotation index.

3.3. `uima-shell`: Integrating non native UIMA third party tools

`uima-shell`[23] offers a way to process Shell command over a CAS element, view or annotation, and to store the result either as a new view or annotation. It mainly aims at running within a UIMA workflow some external third party tools available via command line. These tools must perform their processing by taking the input as a file name parameter or a standard input (*stdin*) and produce the result via the standard output (*stdout*).
The specified CAS element to analyse will be turned into a file argument which will be accessed by specifying the commands and the argument tokens which precede the file argument (i.e. `PreCommand` parameter) as well as the commands and the argument tokens which follow the file argument (i.e. `PostCommand` parameter).
Under a LINUX system, if the command to process takes its input from the standard input, then the `PreCommand` parameter will be set with a `cat` value and the `PostCommand` parameter will starts with a pipe character « `|` » followed by the command. If the command to process takes its input as an argument at a specific position then the `PreCommand` parameter will be set with the command and the first arguments and the `PostCommand` parameter with the last arguments. In any case it is possible to set several commands pre and post the file argument. It is also possible to specify environment variables (i.e. `EnvironmentVariable` parameter) which will be available for the process running the commands.
With the `uima-connectors` project (See Section 3.4.), this current component aims at solving interoperability issues when dealing with non native UIMA tools. The implementation is based on the Martini'Shell API[24].

`http://art.uniroma2.it`

[22]`tmwiki.informatik.uni-wuerzburg.de`

[23]`http://code.google.com/p/uima-shell`

[24]`http://blog.developpez.com/`

3.4. `uima-connectors`: Connecting various text markup languages with UIMA

`uima-connectors`[25] mainly aims at offering solutions to build the bridge between some markup languages and the UIMA CAS.
The Apache UIMA *Tika* [16] project aims at detecting and extracting metadata and structured text content from various type MIME documents. In comparison, `uima-connectors` is more dedicated to perfom mapping from/to text formats to/from CAS, providing solutions for handling general markup language such as eXtended Markup Language (XML), Comma Separated Value (CSV), whitespace-tokenized texts with a sentence per line... or applications of these formats such as Message Understanding Conferences (MUC), Apache OpenNLP, CONLL (B-I-E)...
Similar facilities are offered by other plateforms such as U-Compare, the GrAF softwares[26], GATE. The aim of `uima-connectors` is to offer the most generic solution as possible in order to prevent from having to develop ad hoc solutions or to extend existing ones. The basic idea is to assume that there are recurrent situations where the metamodels of these formats (XML, CSV...) can be aligned with the CAS.
In practice, solutions could be collection readers, analysis engines and CAS consumers. We preferentially adopt an approach in terms of development of AEs which allows to cut into any point of a workflow by specifying the view to process. These components can complete the wrapping performed by the `uima-shell` component but they can also be used at the beginning or at the end of a processing workflow to import or export from/to specific serialization formats.
Examples of `uima-connectors` include the `CSV2CAS` and the `XML2CAS` AEs. The `CSV2CAS` AE offers various ways to create or to update annotations with CSV-like formatted information. The type of the annotation to handle is given as parameter. If the annotation type exists in a specified view, the AE will update the annotations. The number of CSV lines is so assumed to be the same as the number of the annotations instances to update. The AE allows to set the correspondence between features names and column ranks.
The `XML2CAS` AE transduces the information from the XML tree structure to the CAS by creating annotations over the text spans delimited by the begin and the end tags of XML elements. The types of the created annotations are predefined and represent the XML element and attribute nodes: `XMLElementAnnotation` and `XMLAttributeAnnotation`. The `XMLAttributeAnnotation` type, for example, comes with dedicated features to inform about its name (i.e. `attributeName` feature), its value (i.e. `attributeValue` feature) and the element name to which it belongs (i.e. `elementName` feature). As

`adiguba/p3035/java/5-0-tiger/runtime-exec-n-est-pas-des-plus-simple`

[25]`http://code.google.com/p/uima-connectors`

[26]`http://www.americannationalcorpus.org/tools/index.html#uima`

annotations, both have begin, end and coveredText features. This AE offers a way to process the XML structure of any XML document without having to define new UIMA types. In practice, we use the uima-mapper afterwards, to turn the generic types into the specific input annotation types of another following AEs.

4. Examples of use cases

Below we present two real uses cases where we used the components presented in the previous section.

4.1. Mapping annotations: From FTB to HMMPOSTaggerTrainer

In this section we present a situation where it is useful to use a mapping AE to connect two AEs which handles distinct annotations for the same concept. We illustrate this use case on the task of building a statistic model for a Tagger system from the French Treebank (FTB) corpus (Abeillé and Barrier, 2004). We use the Apache HMM tagger trainer [16] (apache-addons:HMMPOSTaggerTrainer) to build an HMM model from the data. The workflow is represented in Figure 2.

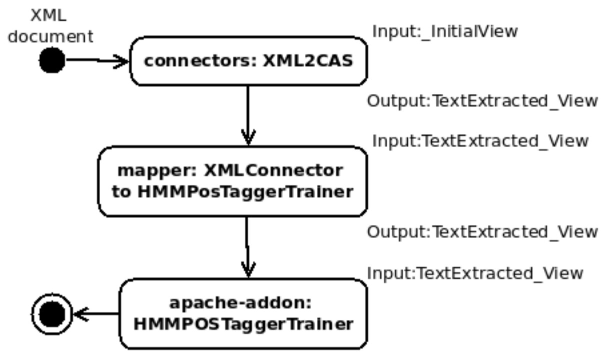
XML document
connectors: XML2CAS
Input:_InitialView
Output:TextExtracted_View
Input:TextExtracted_View
mapper: XMLConnector to HMMPosTaggerTrainer
Output:TextExtracted_View
Input:TextExtracted_View
apache-addon: HMMPOSTaggerTrainer

Figure 2: Mapping annotations: From FTB to HMM-POSTaggerTrainer.

The FTB is available for research purpose in an XML format. Figure 3 shows an example of annotations at various analysis levels. For this use case, we are interested by the XML w element which marks the words. The XML cat, lemma and mph attributes describe respectively the POS, the lemma and some morphological information about the words. The w element is used for simple words and multiword expressions; Embedded w elements do not have a cat attribute but a catint attribute (See the first occurrence of the w element).
The uima-connectors's XML2CAS AE allows to process XML content and to produce annotations for the specified XML nodes. As described in Section 3.4., this AE defines its own TS. Indeed, for each word XML annotation, the AE will create a CAS annotation XMLElementAnnotation for the element w and an XMLAttributeAnnotation for each one of the attributes. For instance, the following XML annotation <w cat="A" mph="A-qual-fp" lemma="petit">petites</w> will produce three XMLAttributeAnnotation annotations; Detail for

```
<SENT nb="8000">
 <w cat="ADV" mph="ADV" lemma="tout au plus">
  <w catint="ADV">Tout</w> <w catint="P">au</w>
  <w catint="D"/> <w catint="ADV">plus</w>
 </w>
 <NP>
  <w lemma="un" cat="D" mph="D-ind-fp">des</w>
  <w cat="A" mph="A-qual-fp" lemma="petit">petites</w>
  <w cat="N" mph="N-C-fp" lemma="chose">choses</w>
 </NP>
 <VPinf>
  <w cat="P" mph="P" lemma="à">à</w>
  <VN>
   <w cat="V" mph="V--W" lemma="changer">changer</w>
  </VN>
  <PP>
   <w cat="P" mph="P" lemma="sur">sur</w>
   <NP>
    <w cat="D" mph="D-def-fs" lemma="le">l'</w>
    <w cat="N" mph="N-C-fs"
      lemma="intégration">intégration</w>
   </NP>
  </PP>
 </VPinf>
 <w cat="PONCT" mph="PONCT-S" lemma=".">.</w>
</SENT>
```

Figure 3: Example of annotations from the lmf3_08000_08499ep.xd.cat.xml file of the FTB. Some attributes have been removed or renamed for readability purpose.

the XMLElementAnnotation is not given here. The table 1 shows the feature names (first line) and the corresponding values of each of the three created XMLAttributeAnnotation annotations. begin and end features and values are not shown to make the table simpler. All the three annotations share the same begin and end values.

elementName	attributeName	attributeValue	coveredText
w	cat	A	petites
w	lemma	petit	petites
w	mph	A-qual-fp	petites

Table 1: Features names and values of the three XMLAttributeAnnotation annotations created for the attributes of the XML annotation <w cat="A" mph="A-qual-fp" lemma="petit">petites</w>. The begin and end features and values are not shown for readability purpose.

For our training purpose let consider we want to work with the w elements which are simple words or parts of a multiword but not a multiword. We are so interested by the XMLAttributeAnnotation annotations which have the following criteria: Either cat or catint as value of the attributeName feature and w as value of the elementName feature. In addition, in order to distinguish the XMLAttributeAnnotation annotations which correspond to simple words from those which correspond to embedding w elements (both have cat as value of the attributeName feature), we decide to filter them out based on the presence of a whitespace character in the covered text. Figure 4 shows an example of uima-mapper's rule for mapping annotations which implements this definition. The rule declares that for each XMLAttributeAnnotation annotation, if the XPath

```
<rule id="XMLAttributeAnnotationW2ApacheTokenAnnotation"
 description="In the FTB, select the w elements which are simple words or parts of a multiword but not one">
 <pattern>
  <patternElement type="fr.univnantes.lina.uima.connectors.types.XMLAttributeAnnotation">
   <constraint>.[(@elementName='w') and ((@attributeName='cat') or (@attributeName='catint'))
                and not(contains(@coveredText,' '))]</constraint>
   <create type="org.apache.uima.TokenAnnotation" >
    <setFeature name="posTag" value="normalize-space(./@attributeValue)"/>
   </create>
  </patternElement>
 </pattern>
</rule>
```

Figure 4: Example of an `uima-mapper`'s rule for mapping UIMA annotations.

`constraint` is satisfied, then it will imply the creation of a `TokenAnnotation` at the same offsets and set its `posTag` feature with a value resulting from another XPath processing on the matched annotation. This example gives an idea of the expressive power of XPath which allows to express constraints with many kinds of operators (e.g. boolean, comparison, regular expression…) and functions (e.g. string, mathematical…).

4.2. Integrating a third party tool: TreeTagger

In this section we show how to process UIMA CAS elements with command line tools and how to set the UIMA CAS elements with the produced results. In particular we describe how to integrate a command line tool with CSV-like input and output formats. We illustrate this use case by setting the POS and the lemma features of some CAS token annotations with the processing results of an external POS tagger.

As a third party tool example, we use the (Schmid, 1994)'s POS tagger (also named «TreeTagger») which takes a word per line and produces POS and lemma annotations as tabulated values aside of each input word (i.e. `petites ADJ petit`). Both input and output formats can be considered as CSV-like formats; The input format being a special case with a single column. We also use the Apache whitespace tokenizer [16] (`apache-addons:wst`) which segments text into word token annotations.

The workflow is represented in Figure 5. Each component works on a specified input view and stores its result in a specified output view. The raw text of a document is present in the view `_InitialView` at the beginning of the processing. The `CAS2CSV` and the `CSV2CAS` AEs are part of the `uima-connectors` project. They take in charge the conversion from/to CSV-like formats to/from UIMA CAS elements. The `shell` AE refers to the `uima-shell` AE and allows the integration of TreeTagger.

The processing proceeds like this: The Apache whitespace tokenizer performs the segmentation and stores the token annotations (i.e. `TokenAnnotation`) in its working input view `_InitialView`. The POS tag and the lemma features are not yet set. The `CAS2CSV` AE takes in parameters the names of an annotation type (i.e. `TokenAnnotation`) and of the features (i.e. `coveredText`) to process. For each annotation instances present in the given `_InitialView`, it creates a CSV-formatted line with the values of the specified features at a specified column rank. The result is stored in the `CAS2CSV_View`. The

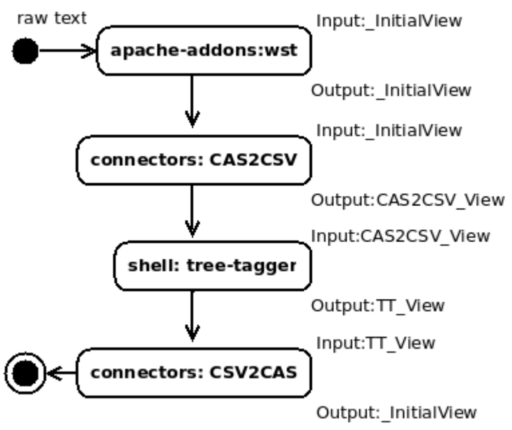

Figure 5: Integrating a third party tool: TreeTagger.

`uima-shell` AE is set with the following parameters values: The `EnvironmentVariable` parameter declares `TT_HOME=/path/to/application/home/tree-tagger`, and the `PreCommand` and the `PostCommand` parameters declare respectively the following values: `cat` and `| ${TT_HOME}/bin/tree-tagger ${TT_HOME}/lib/french.par -token -lemma -sgml`. The `CSV2CAS` AE parses the `TT_View`. Thanks to the given name of an annotation type (i.e. `TokenAnnotation`) and the associations of column ranks with feature names specified by parameters (e.g. `1 -> posTag ; 2 -> lemma`), the process updates each annotation instance from the given `_InitialView` with information coming from the CSV-formatted content; Each line corresponds to an annotation. The POS tag and the lemma features are so set.

It is interesting to note that the `apache-addons:wst` and the `CAS2CSV` components can simply be removed and functionally replaced by a command line token such as `| perl -ne 'chomp; s/(\p{IsAlnum})(\p{IsPunct})/$1 $2/g; s/(\p{IsPunct})(\p{IsAlnum})/$1 $2/g; s/ /\n/g; print'` in the `PostCommand` parameter before the TreeTagger command.

5. Conclusion

This `uima-common` project is a common basis for all the uima-components we develop such as `uima-mapper`, `uima-shell`, and `uima-connectors`. It was also used for example for developing a wrapper for the java implementations of the C99 and TextTiling segmentation algorithms, written by Freddy Choi `uima-text-segmenter`[27].

All the presented components aims at tackling the problem of interoperability within a UIMA workflow. They were both developed and used for building workflows in particular for importing and connecting an XML version of the French Treebank corpus (Abeillé and Barrier, 2004) to trainer systems such as the Apache HMM tagger[28] and the OpenNLP MaxEnt preliminary processing tools (Boudin and Hernandez, 2012).

As perspectives, we want primarily to inform about these solutions to motivate people to use them and to participate to the development of these projects. Considering our short term developments, we plan to concentrate our efforts on the `uima-mapper`. We are thinking in some ways to build UIMA annotations index which could effectively support the implementation of recognition mechanisms for matching annotations patterns. We also would like to evaluate the performance of the various existing solutions in terms of expressiveness and resource consuming.

The `uima-connectors` will also evolve. As we previously mentioned, we are more interested by generic approaches than ad hoc ones. Currently, we offer a solution to import to the CAS, the information that can be inferred from the tree structure of XML formats (e.g. two elements can be considered as in relation due to the fact that one of them encloses the other one). Since, several XML formats, such as GrAF, are used to represent graph structures thanks to common mechanisms (e.g. couples of id/idref attributes), we are interested to offer a way to specify generically these mechanisms and to align them with elements of the CAS structure in order to be able to handle any XML documents representing graphs.

Concerning the `uima-shell`, the current version does not address the security problems such as the injections of malicious codes or the lack of system resources to perform safely the commands. Indeed, the commands, which are intended to be run, are assumed to be performed by an allowed user as well as to be safe for the running system. We believe that the component should not be restricted because it runs counter to what it aims to do. And in addition, we believe that it can not be restricted because it is impossible to *a priori* enumerate all the unsafe use cases. The best solutions to prevent these risks are to run the vulnerable process in «jail» environments with restricted rights and resources (i.e. to configure correctly the embedding application servers and operating systems).

6. Acknowledgements

This work was financially supported by *La région Pays de la Loire* under the DEPART project `http://www.projet-depart.org`.

I also would like to express my gratitude to the reviewers for there insightful comments.

[27]`https://code.google.com/p/uima-text-segmenter`

[28]`http://enicolashernandez.blogspot.com/2011/05/construire-des-modelisations-du-french.html`

7. References

Anne Abeillé and Nicolas Barrier. 2004. Enriching a french treebank. In *Actes de la conférence LREC*, Lisbonne.

Florian Boudin and Nicolas Hernandez. 2012. Détection et correction automatique d'erreurs d'annotation morpho-syntaxique du french treebank. In *TALN*.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A. Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. 2011. Text processing with gate. University of Sheffield Department of Computer Science. ISBN 0956599311, April 15th.

Hamish Cunningham. 2002. Gate, a general architecture for text engineering. *Computers and the Humanities*, 36:223–254.

David Ferrucci and Adam Lally. 2004. Uima: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348.

Iryna Gurevych, Max Mühlhäuser, Christof Müller, Jürgen Steimle, Markus Weimer, and Torsten Zesch and. 2007. Darmstadt knowledge processing repository based on uima. In *Proceedings of the First Workshop on Unstructured Information Management Architecture at Biannual Conference of the Society for Computational Linguistics and Language Technology*, Tübingen, Germany.

Udo Hahn, Ekaterina Buyko, Katrin Tomanek, Scott Piao, John McNaught, Yoshimasa Tsuruoka, and Sophia Ananiadou. 2007. An annotation type system for a data-driven nlp pipeline. In *The LAW at ACL 2007 – Proceedings of the Linguistic Annotation Workshop*, pages 33–40. Prague, Czech Republic, June 28-29, 2007. Stroudsburg, PA: Association for Computational Linguistics.

Nicolas Hernandez, Fabien Poulard, Matthieu Vernier, and Jérôme Rocheteau. 2010. Building a French-speaking community around UIMA, gathering research, education and industrial partners, mainly in Natural Language Processing and Speech Recognizing domains. Proceedings of the LREC 2008 Workshop 'New Challenges for NLP Frameworks', La Valleta, Malta, 05.

Nancy Ide and Keith Suderman. 2009. Bridging the gaps: interoperability for graf, gate, and uima. In *Proceedings of the Third Linguistic Annotation Workshop*, ACL-IJCNLP'09, pages 27–34, Stroudsburg, PA, USA. Association for Computational Linguistics.

Yoshinobu Kano, Luke McCrohon, Sophia Ananiadou, and Jun'ichi Tsujii. 2009. Integrated NLP evaluation system for pluggable evaluation metrics with extensive interoperable toolkit. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing (SETQA-NLP 2009)*, pages 22–30, Boulder, Colorado, June. Association for Computational Linguistics.

Peter Kluegl, Martin Atzmueller, and Frank Puppe. 2009. Textmarker: A tool for rule-based information extraction. In Christian Chiarcos, Richard Eckart de Castilho, and Manfred Stede, editors, *Proceedings of the Biennial GSCL Conference 2009, 2nd UIMA@GSCL Workshop*, pages 233–240. Gunter Narr Verlag.

Frédéric Meunier, Philippe Laval, Gaëlle Recourcé, and Sylvain Surcin. 2009. Kwaga : une chaîne uima d'analyse de contenu des mails. In *First French-speaking meeting around the framework Apache UIMA at the 10th Libre Software Meeting*, University of Nantes, July.

Sébastien Paumier. 2003. A Time-Efficient Token Representation for Parsers. pages 83–90. Proceedings of the EACL Workshop on Finite-State Methods in Natural Language Processing.

Laurent Romary and Nancy Ide. 2004. International Standard for a Linguistic Annotation Framework. *Natural Language Engineering*, 10(3-4):211–225, September.

Helmut Schmid. 1994. Probabilistic part-of-speech tagging using decision trees. In *Proceedings of the Conference on New Methods in Language Processing*, Manchester, UK.

Dhaval Thakker, Taha Osman, and Phil Lakin. 2009. Gate jape grammar tutorial, February 27.

Paul Thompson, Yoshinobu Kano, John McNaught, Steve Pettifer, Teresa Attwood, John Keane, and Sophia Ananiadou. 2011. Promoting interoperability of resources in meta-share. In *Proceedings of the Workshop on Language Resources, Technology and Services in the Sharing Paradigm*, pages 50–58, Chiang Mai, Thailand, November. Asian Federation of Natural Language Processing.

Karin Verspoor, William Baumgartner Jr., Christophe Roeder, and Lawrence Hunter. 2009. Abstracting the types away from a uima type system. In *2nd UIMA Workshop at Gesellschaft für Sprachtechnologie und Computerlinguistik (GSCL)*, Tagung, Germany, October.