Procesamiento del Lenguaje Natural, Revista nº 52 marzo de 2014, pp 21-28 recibido 14-11-13 revisado 05-02-14 aceptado 06-02-14

Fénix: a flexible information exchange data model for natural language processing*

Fénix: un modelo de datos flexible para el intercambio de información en procesamiento del lenguaje natural

José M. Gómez, David Tomás, Paloma Moreda
Depto. de Lenguajes y Sistemas Informáticos - Universidad de Alicante
Carretera San Vicente del Raspeig s/n - 03690 Alicante (Spain)
{jmgomez,dtomas,moreda}@dlsi.ua.es

Resumen: En este artículo se describe Fénix, un modelo de datos para el intercambio de información entre aplicaciones en el campo del Procesamiento del Lenguaje Natural. El formato propuesto está pensado para ser lo suficientemente flexible como para dar cobertura a estructuras de datos, tanto presentes como futuras, empleadas en el campo de la Lingüística Computacional. La arquitectura Fénix está dividida en cuatro capas: conceptual, lógica, persistencia y física. Esta división proporciona una interfaz sencilla para abstraer a los usuarios de los detalles de implementación de bajo nivel, como los lenguajes de programación o el almacenamiento de datos empleado, permitiéndoles centrarse en los conceptos y procesos a modelar. La arquitectura Fénix viene acompañada por un conjunto de librerías de programación para facilitar el acceso y manipulación de las estructuras creadas en este marco de trabajo. También mostraremos cómo se ha aplicado de manera exitosa esta arquitectura en diferentes proyectos de investigación.
Palabras clave: modelo de datos, herramientas de PLN, integración de recursos, intercambio de información

Abstract: In this paper we describe Fénix, a data model for exchanging information between Natural Language Processing applications. The format proposed is intended to be flexible enough to cover both current and future data structures employed in the field of Computational Linguistics. The Fénix architecture is divided into four separate layers: conceptual, logical, persistence and physical. This division provides a simple interface to abstract the users from low-level implementation details, such as programming languages and data storage employed, allowing them to focus in the concepts and processes to be modelled. The Fénix architecture is accompanied by a set of programming libraries to facilitate the access and manipulation of the structures created in this framework. We will also show how this architecture has been already successfully applied in different research projects.
Keywords: data model, NLP tools, resource integration, information exchange

1 Introduction

Any research work should be motivated by the idea of sharing knowledge, tools and resources that can be employed by other researchers to jointly improve their area of expertise. The research carried out in the field of Natural Language Processing (NLP) relies heavily on resources and tools previously developed by other community members. For instance, a text classification system may depend on the output generated by morphological tools (e.g., part-of-speech taggers), syntactic tools (e.g., shallow parsers), and semantic tools (e.g., named entity recognizers). If this system followed a machine learning-based approach, it could also require as an input a corpus to train and validate the system.

At some point in the process of developing almost any NLP application, every researcher faces the problem of integrating diffe-

* This research has been partially funded by the Spanish Ministry of Economy and Competitiveness under project LegoLangUAge (Técnicas de Deconstrucción en las Tecnologías del Lenguaje Humano, TIN2012-31224).

ISSN 1135-5948

rent tools and resources in their frameworks. In these situations, researches and developers usually have to do a significant effort to adapt and integrate their products with previously existing ones, since different people employ different input and output formats. Moreover, this effort has to be done any time a component is changed by a different one. In a worst case scenario where n different inputs for m different tools are available, a total of $n{\cdot}m$ conversions between formats must be done in order to process them all. This problem could be mitigated by establishing a common information exchange format, adapting the n different inputs to this new format. We could also adapt the m different tools to process this common format and reduce the need for conversions to just $n + m$ different possibilities in this case.

The problem of integrating tools and resources not just only affects the cooperation between different research groups, but also intra-group collaboration suffers from that problem (Moreno-Monteagudo and Suárez, 2005). Taking into account the amount of tools and resources available nowadays, it becomes increasingly necessary the development of frameworks to easily integrate heterogeneous sources of information to build up more complex NLP systems. Moreover, standardizing inputs and outputs not just facilitates researchers the consumption of resources and tools, but also the dissemination of their work and its citing and reuse by other community members.

In this paper we present Fénix, an information exchange data model to facilitate sharing of information between different NLP processes. The purpose of this model is to provide a standard to encode inputs and outputs for different process types in the field of computational linguistics (part-of-speech taggers, syntactic parsers, text classifiers, etc.), facilitating in this way the integration of different NLP resources and tools. Although this paper focuses in the application of Fénix in the area of NLP, the model is flexible enough to be applied in any process communication context.

Our proposal tries to bring together the most relevant features included in previous models, covering the gaps in existing work. The most relevant feature of Fénix, which distinguishes it from other approaches, is the adaptability. The model proposed is not limited to a fixed set of predefined types, since new data types can be defined for new processes as necessary. This flexibility does not have an impact in the usability of the model, since Fénix provides a simple interface based on a four layer architecture that abstracts the user from implementation details, such as data structures and storage.

The remainder of this article is organized as follows. Next section reviews the related work in the field of NLP processes communication and integration. Section 3 describes all the components involved in the Fénix architecture. Section 4 provides details on implementation experiences already carried out with Fénix in different research projects. Finally, Section 5 summarises conclusions and future work.

2 Related Work

There are two main approaches in the existing research works carried out for the integration of NLP resources and tools: (i) projects that only define the data format used by processes to communicate between them; (ii) projects that take into account data and tools in a unique platform.

Regarding data integration, we can highlight the Annotation Graph Toolkit (Maeda et al., 2001) and the Atlas architecture (Bird et al., 2000). Both systems propose a three level architecture, comprising logical, physical, and application levels. The logical level implements a generalization of the annotation graph model presented by Bird and Liberman (2001). Although this logical level provides independence of the application and the physical storage, it does not allow separating the information in different layers. Thus, every process has to upload all the previous annotations to complete the task.

Another relevant system is EMU (Cassidy and Harrington, 2001), a system intended for labelling, managing, and retrieving data from speech databases. Although EMU is portable to major computing platforms and provides integration of hierarchical and sequential labelling, the area of application is limited to speech data.

With respect to data and tools integration, two systems have been widely employed by the NLP community: GATE (Cunningham et al., 2011) and UIMA (Ferrucci and Lally, 2004). The first one offers a framework and an environment for language engineering.

As a framework, it provides a set of software components that can be used, extended, and customised for specific needs. As a development environment, it facilitates adding new components. The process of integrating new components is straightforward in the case of Java. However, for other programming languages this process is more complicated since each resource is treated as a Java class.

On the other hand, UIMA is the result of the efforts carried out by IBM to create a common architecture and a robust software framework that would be able to reuse and combine results of its different working teams, accelerating the process of transferring the advances made in NLP into the IBM's product platform. Although it provides a common framework for combining NLP components, these components are always limited to IBM products.

Thus, although some efforts have been made to develop integration platforms in the field of NLP, none of them is flexible and general enough to provide a definite, easy, and adaptable information exchange model to the NLP community. In this sense, the proposal described in this paper provides a simple interface based on a four-layer architecture, comprising conceptual, logical, persistence, and physical levels. Previous layer-based models usually define three layers, jointly considering physical and persistence layers. This distinction in our proposal allows the storage of information in different formats by just modifying the persistence layer (see Section 3).

Another relevant feature of Fénix is the possibility of distributing the information in different sources. For instance, the result of a part-of-speech (POS) tagger could be stored in a file, whereas the original text could remain in a different file, providing links between the initial tokens and the POS labels assigned. In this way, unlike many previous approaches, it is not necessary to load all the information for every process, focusing only on the data necessary to accomplish a particular task.

Fénix was originally conceived as part of the InTime architecture (Gómez, 2008), an integration platform for NLP tools and resources. In this platform, Fénix provides the data model to facilitate the information exchange between heterogeneous processes. As part of this architecture, there is a set of libraries available for developers to create, access, and modify Fénix objects.

3 Fénix Architecture

Fénix is a data model for information exchange between computational linguistics processes. Due to the heterogeneity of systems and tasks in this research area, it is very difficult to define all the possible types of data structures that may be necessary in this field. That is why Fénix's philosophy is based on a logic model flexible enough to incorporate both current and possible future data structures. In order to achieve this goal, Fénix is divided into four separate layers: conceptual, logical, persistence, and physical. Figure 1 shows how different layers are related in our model.

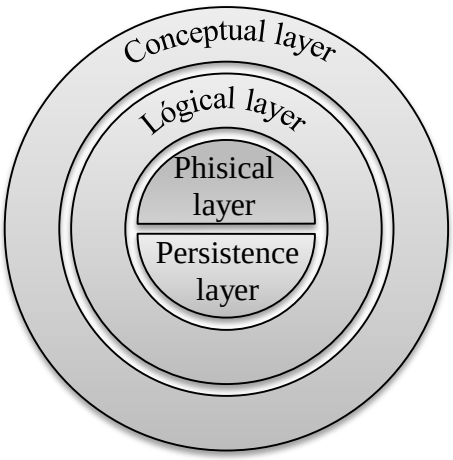

Figura 1: Fénix four-layer architecture.

The conceptual layer is in the top level and it is used to define the conceptual objects, called *object wrappers* in Fénix. These objects will provide the input and output public interfaces in order to abstract the logical layer and its structure to the end user. For instance, if we add a text string into Fénix, the end user will use a wrapper interface to interact with it. In this case, a Fénix object is created, which represents an instance of the text concept in the model. The interfaces of this object will have the necessary public methods, for instance *getText()* and *setText()*, to access from or store into Fénix text objects. Each type of wrapper has its own interface and involves different model concepts. For example, an input text, a classification result, tokens from a given text, or a search result will be considered different concepts, and thus different wrapper types in the Fénix model.

The conceptual layer is based on the logical layer, which defines complex information elements and their structure. The logical layer consists of information elements called `unit`. These elements are indivisible and represent the result of a process, where a process can generate more than one information `unit`. Each `unit` represents a type of data that could be considered simple (e.g., a string) or complex (e.g., the result of a text classification or an information search), containing a type that reveals its structure and what information is included. That is, `unit` elements of the same type have the same structure. For example, the `unit` type `plain_text` could store a text string, an optional source, and the start and final position of the text (the relative position from the beginning of the document where the string was located). The source is a reference to a related set `unit` elements, indicating from what `unit` elements was the information obtained. For instance, we could obtain a `text_plain` element without stopwords from the original text which included these terms.

The persistence and physical layers are in the lowest level. The physical layer defines how the Fénix model is implemented and which programming languages can be employed to process the model. On the other hand, the persistence layer defines how to import and export each Fénix object and in what formats can the data be persistent. In fact, different objects can be stored in different formats, also offering the possibility of distributing the information on disk and memory to optimize the use in several tasks. Moreover, the user can decide which objects are finally stored and which are not.

Figure 2 shows the structure of the different modules of Fénix model and its components. For clarity, the physical layer is not shown in this scheme, but it is the basis to implement the entire structure of the model.

A `unit` is composed of one or more complex information structures called `item`. This `item` represents a part of the information contained in a `unit`, but they become useful only when considered together with other `item` elements of the `unit`. An XML representation of this model is shown as follows:

```
<fenix version="1.0.0">
  <unit id="unit_id" type="unit_type"
      [tool="tool_name"]>
    <item id="item_id_1" data_type="simple">
      <info id="id_1.1" data_type="info_type">
        value
      </info>
    </item>
    <item id="item_id_2" data_type="vector">
      <item id="0" data_type="item_type">
        ...
      </item>
      <info id="1" data_type="info_type">
        value
      </info>
      <item id="2" data_type="item_type">
        ...
      </item>
      <info id="3" data_type="info_type">
        value
      </info>
      ...
    </item>
    <item id="item_id_3" data_type="struct">
      <item id="id_3.1" data_type="item_type">
        ...
      </item>
      <info id="id_3.2" data_type="info_type">
        value
      </info>
      <item id="id_3.3" data_type="item_type">
        ...
      </item>
      <info id="id_3.4" data_type="info_type">
        value
      </info>
      ...
    </item>
    ...
  </unit>
  ...
</fenix>
```

The `unit` has the attribute `tool` that indicates from which tool has been obtained the data of this `unit`. This attribute is optional and cannot be set if it is unknown or the `unit` represents the input data. All `unit`, `item` and `info` elements contain an identifier. Whereas the `unit` identifier must be unique (cannot exist two `unit` with the same identifier), the `item` or `info` identifier must be unique only at the level of its container (a `unit` or another `item`). For example, the following code represents the output of a *question answering* system that returns three information units: the input question, the question language detected by the system, and the answers found.

```
<fenix version="1.0.0">
<unit id="input_question" type="plain_text">
  <item id="text" data_type="simple">
    <info id="value" data_type="string">Who is the
        president of Spain?</info>
  </item>
</unit>
<unit id="question_lang" type="lang" tool="jirs">
  <item id="sources" data_type="struct">
    <info id="text" data_type="id">input_query</
        info>
  </item>
  <item id="lang" data_type="simple">
    <info id="value" data_type="string">en</info>
  </item>
```

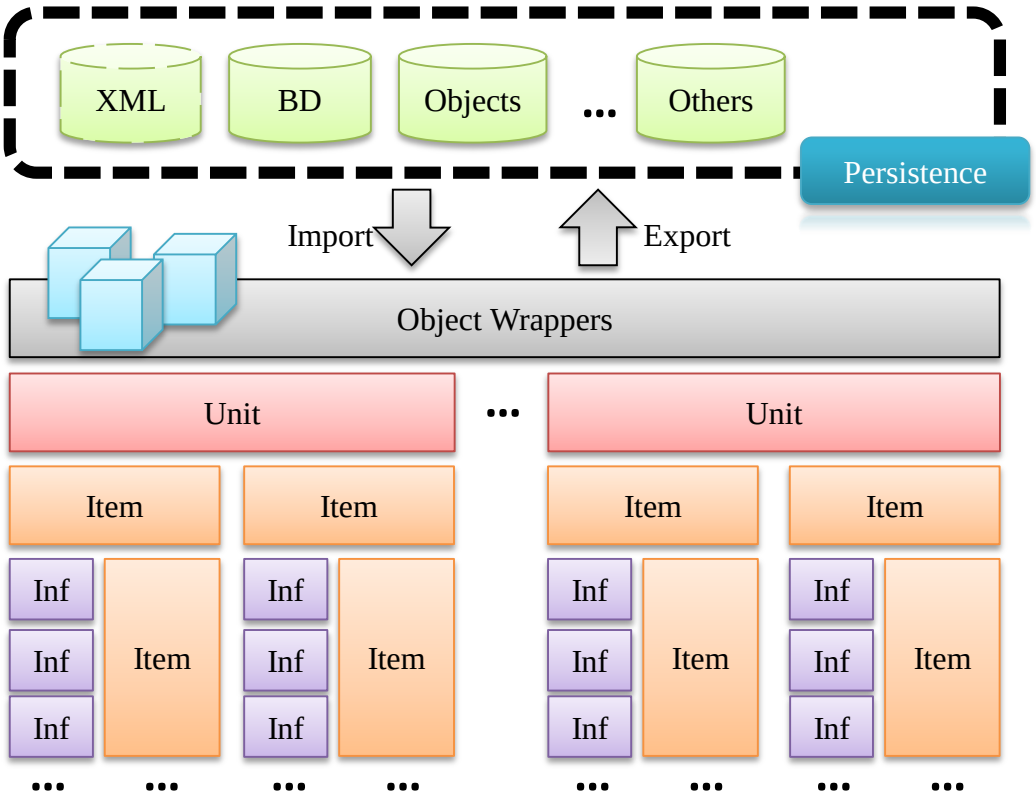

Figura 2: Module structure in Fénix.

```
</unit>
<unit id="answers" type="answers" tool="jirs">
  <item id="sources" data_type="struct">
    <info id="question" data_type="id">
        input_question</info>
    <info id="lang" data_type="id">question_lang</
        info>
  </item>
  <item id="results" data_type="vector">
    <item id="0" data_type="struct">
      <info id="text" data_type="string">Mariano
          Rajoy</info>
      <info id="score" data_type="float">1.0</info
          >
    </item>
    <item id="1" data_type="struct">
      <info id="text" data_type="string">Jos Luis
          Rodrguez Zapatero</info>
      <info id="score" data_type="float">0.8</info
          >
    </item>
  </item>
</unit>
</fenix>
```

category in the Fénix model. It should be noted that these are only a small sample of the established `unit` types, and the model is open to include new types with its own internal structure and wrappers.

As we can see in the previous example, all the identifiers (`id`) of a `unit` are different. Nevertheless, the item `sources` appears in `question_lang` and `answers` entries, whereas `item` identifiers `text` and `score` occur in several subitems of `results`. Although any `id` is repeated in the same scope, cannot exist neither two item `sources` as children of the unit `answers`, nor two info `text` as children of the item `answers.results.1`. The label `unit` can be assigned many different types and it is open to new types of information to be incorporated in the future. Whenever a new `unit` type is created, the XML specification is added in the project Wiki.[1] Some `unit` types already implemented are: `plain_text`, for plain texts; `categories`, to store different categories for a classification process; and `classification` which relates a sample with its category in the Fénix model. It should be noted that these are only a small sample of the established `unit` types, and the model is open to include new types with its own internal structure and wrappers.

The `item` data type, however, can only be one of three different types: `simple`, `vector`,

[1] http://intime.dlsi.ua.es/fenix/.

and `struct`. An `item` of type `simple` contains only one `info` element; an `item` of type `vector` contains a sequence of `item` or `info`; finally, an `item` of type `struct` contains a complex structure formed by other `item` or `info` elements, which could be referenced by an identifier `id`. As shown in the previous example, `item` may contain other `item` elements or basic format information `info`, the terminal nodes of the model. Therefore, information units may be composed of various combinations of `item` and `info` elements, considering two limitations: `info` elements must have an `item` parent and should be the terminal nodes of the model, i.e., they must contain a basic data type and cannot include other `item` or `info` elements. The `info` elements can only pertain to one of the following basic types:

`character`: individual characters

`string`: sequence of characters

`integer`: integer value without decimal part

`float`: simple precision floating point number

`double`: double precision floating point number

`date`: date/time in different formats depending on location

`object`: programming object

`id`: reference to another Fénix element

It is worth noting the `object` and `id` data types. Since Fénix, apart from a model is a framework for data exchanging between different processes, we considered necessary to allow the storage of programming objects in the `info` elements. For instance, we could store in a process the database connection as a JAVA object, passing it to another process instead of opening a new database connection every time.

The last basic type of Fénix information is `id`. All Fénix elements (`unit`, `item` and `info`) have an identifier and can be referenced by an `info` element of type `id`. These identifiers are hierarchical, being `unit` the top level element that can be referenced, and `info` the lowest one. Therefore, references of type `id` are formed by concatenating the identifier of all parent nodes to the element you want to reference, separated by dots. Consider the following example:

```
<unit id="search_result" type="snippets" [tool="
    tool_name"]>
  <item id="sources" data_type="struct">
    <info id="query" data_type="id">input_query</
        info>
  </item>
  <item id="results" data_type="vector">
    <item id="0" data_type="struct">
      <info id="url" data_type="string">url_1</
          info>
      <info id="title" data_type="string">title_1<
          /info>
      <info id="snippet" data_type="string">
          snippet_1</info>
      <info id="score" data_type="float">score_1</
          info>
    </item>
    <item id="1" data_type="struct">
      <info id="url" data_type="string">url_2</
          info>
      <info id="title" data_type="string">title_2<
          /info>
      <info id="snippet" data_type="string">
          snippet_2</info>
      <info id="score" data_type="float">score_2</
          info>
    </item>
    ...
  </item>
</unit>
```

The `unit` type `snippets` is employed for storing the results of a search engine. If we would like to reference the URL of the second snippet, the identifier will be `search_result.results.1.url`, where `search_result` is the information unit identifier, `results` is the identifier of the second item which contains the snippet list, `1` is the item number inside of the result vector `results`, and `url` is the `info` element with the information to retrieve.

In the previous example, an `item` with the identifier `sources` is also present. This special `item` can occur in any information unit and it is employed to know from which information unit or units it was obtained. Following the previous example, thanks to this `item` we know that the search result has been obtained from a query whose text is in the `unit` with the `input_query` identifier. Thus, if needed, we can track back the information to the source and retrieve intermediate results which would otherwise be missed. For example, if we model a POS tagger, the result of this tool would be a list of values that correspond to each POS tag from the initial text. But if we want to display the final result (the POS tags) and the original terms together, we could use these backward references.

4 Applications

Fénix has already been successfully applied to several projects developed in our research group. The two main applications where this model were used are Java Process Manager[2] (JPM) and InTime.[3]

structure of Project 1 was notably simplified, allowing to make simpler and more independent processes.

JPM is a development framework for creating processes in the area of NLP. It is focused on developing modular and customizable tools for researchers to easily test different modules for the same NLP task, just by changing the system parameters. Moreover, JPM allows integrating both native and external processes, independently from the program language or the operating system employed. A JPM application defines its behaviour thanks to a configuration file that indicates which processes are executed, in which order and conditions. One of the main advantages of this framework is the possibility of changing the tool process architecture by just modifying the configuration file. For example, we could convert a pipeline to a client/server architecture, a parallel processing, a distributed processing or a combination of them. One of the most relevant problems of JPM, before the inclusion of the Fénix model, was how to share the data between processes. Thanks to Fénix, the configuration file structure of JPM was notably simplified, allowing to make simpler and more independent processes.

On the other hand, InTime is a distributed integration and exchange platform of tools and resources based on P2P technology. The goal of InTime is to provide researchers with a simple shared platform to discover and use tools developed by other researchers. Employing Fénix was basic in order to provide a shared space of data exchange between processes.

Other applications in which Fénix has been successfully employed are: GPLSI Dossier,[4] an application to classify news based on customers criteria; GPLSI Classifier, a text classifier based on different NLP processing tools (tokenization, lemmatization, n-gram extraction, etc.); MONEI,[5] a meta-search engine for business opportunities in foreign markets; Pyramid, an Internet crawler which is able to process hundreds of thousands of web pages per day; and Social Observer,[6] an application which monitorizes tweets and gives them a value according to their sentiment polarity.

Finally, Fénix is currently being employed in the LegoLangUAge[7] project, as the basis to build up the basic information units called *L-Bricks*. These units define the data structures and their relations between other L-Bricks and the ontology of the system. Fénix was chosen among other data models due to its flexibility and coverage to all the needs of the LegoLangUAge project.

Regarding the application of Fénix to these projects, building Fénix objects was just a matter of building a wrapper based on the basic structures described before: `units`, `structs` and `infos`. Objects created in this way were shared by means of Subversion[8] in a Sourceforge repository,[9] being immediately available for any user. We developed templates for Netbeans[10] providing the basic wrapper's structure and methods to work with it. Developing a wrapper can be performed in less than an hour for someone with a reasonable knowledge of the model. Once created, the further use of the wrapper by other researchers is straightforward.

5 Conclusions and Future Work

In this paper we have presented Fénix, a data model designed for information exchange between NLP processes. The data model proposed is flexible and scalable, intended to provide a generic data representation relying on a reduced set of basic tags to codify a wide coverage of NLP tools and corpus.

The Fénix architecture is accompanied by a set of programming libraries to facilitate the access and manipulation of the structures created in this framework. We have also presented a set of research projects and tools where this architecture has been already successfully applied. The application of Fénix allowed simplifying the integration and communication between processes in all the con-

[2] http://gplsi.dlsi.ua.es/gplsi11/content/jpm-31.

[3] http://gplsi.dlsi.ua.es/gplsi11/content/intime-platform.

[4] http://gplsi.dlsi.ua.es/gplsi11/content/dossier.

[5] http://intime.dlsi.ua.es:8080/monei/.

[6] http://gplsi.dlsi.ua.es/gplsi11/content/gplsi-social-observer.

[7] http://gplsi.dlsi.ua.es/legolang.

[8] http://subversion.tigris.org/.

[9] http://sourceforge.net/.

[10] https://netbeans.org/.

texts described.

As future work, we plan to continue with the application of the model as the core of the information exchange in our current and next developments, with a particular focus on setting the basis in LegoLangUAge to build up the basic information units called *L-Bricks*. We also plan to further release new libraries in different programming languages to facilitate accessing Fénix to the NLP research community.

References

Bird, S., D. Day, J. Garofolo, J. Henderson, C. Laprun, and M. Liberman. 2000. Atlas: A flexible and extensible architecture for linguistic annotation. In *Proceedings of the second international conference on Language Resources and Evaluation*, LREC '00.

Bird, S. and M. Liberman. 2001. A formal framework for linguistic annotation. *Speech Communication*, 33(1-2):23–60.

Cassidy, S. and J. Harrington. 2001. Multilevel annotation in the emu speech database management system. *Speech Communication*, 33(1–2):61–77.

Cunningham, H., D. Maynard, K. Bontcheva, V. Tablan, N. Aswani, I. Roberts, G. Gorrell, A. Funk, A. Robert, D. Damljanovic, T. Heitz, M. A. Greenwood, H. Saggion, J. Petrak, Y. Li, and W. Peters. 2011. *Text Processing with GATE (Version 6).*

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

Gómez, J. M. 2008. Intime: Plataforma de integración de recursos de pln. *Procesamiento del Lenguaje Natural*, 40:83–90.

Maeda, K., S. Bird, X. Ma, and H. Lee. 2001. The annotation graph toolkit: software components for building linguistic annotation tools. In *Proceedings of the first international conference on Human language technology research*, HLT '01, pages 1–6, Stroudsburg, PA, USA. Association for Computational Linguistics.

Moreno-Monteagudo, L. and A. Suárez. 2005. Una propuesta de infraestructura para el procesamiento del lenguaje natural. *Procesamiento del Lenguaje Natural*, 35.