NLP Tools Integration Using a Multi-Layered Repository

João Graça[*], Nuno J. Mamede[*], João D.Pereira[†]

[*]L^2F – INESC-ID Lisboa/IST
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
{joao.graca, nuno.mamede}@l2f.inesc-id.pt
[†] Eng. Group –INESC-ID Lisboa/IST
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
joao@inesc-id.pt

Abstract

Natural Language processing (NLP) systems are typically characterized by a pipeline architecture in which several independently developed NLP tools, connected as a chain of filters, apply successive transformations to the data that flows through the system. Hence when integrating such tools, one may face problems that lead to information losses, such as: (i) tools discard information from their input which will be required by other tools further along the pipeline; (ii) each tool has its own input/output format. Moreover, the tools formats do not establish relations between their Input/Output. These relations are useful for keeping the information of different levels aligned. Another problem is that each tool developer must normally concern himself with the definition of a data model to represent linguistic information and Input/Output facilities for the tool he's developing. Usually, these models and facilities are very similar, so their redefinition for each tool represents a waste of time.

This work proposes a solution that solves these problems. We offer a framework for NLP systems. Our framework uses a client-server architecture, in which the server acts as a repository where all tools add/consult data. Data is kept in the server under a data model that is independent of the client tools. This model is able to represent a broad range of linguistic information. The tools interact with the server through a generic API which allows the creation of new data and the navigation through all the existing data. Moreover, we provide libraries implemented in several programming languages that abstract the connection and communication protocol details between the tools and the server, and provide several levels of functionality that simplify server use.

1. Introduction

Natural Language processing (NLP) systems are typically characterised by a pipeline architecture, in which several NLP tools connected as a chain of filters apply successive transformations to the data that flows through the system. Usually, each tool is independently developed by a different person whose focus is on his/her own problem rather than on the future integration of the tool in a broader system. Hence when integrating such tools, several problems arise, which are mainly related to the following: (i) how the tools communicate with each other, (ii) what kind of information flows between the several tools (may cause information lost).

At the Spoken Language Systems Lab (L^2F), where this work was developed, several NLP systems have already been created. Most of the detected problems when bulding those systems where related with the information flow between the tools composing the system. These problems are:

- Architectural problems - the information discarded along the system may be required further ahead by other tools;
- Conversion between data formats - conversions are necessary between the data formats produced by different tools that wish to interchange information. Moreover, if the expressiveness of each format is different, then one format may not be completely mappable into another.

Besides these problems, which lead to information losses, there is another problem concerning the data: how to maintain the data lineage between information produced by different tools composing a NLP system? When viewing each tool output as a layer of linguistic information over a primary data source and considering that layers are normally related to each other, it is desirable to maintain relations between those layers. First, these relations enable the navigation through related linguistic information produced by different tools. Secondly, tools can reference data from other layers in order to avoid the repetition of common data. These types of relations are called cross-relations because they span across linguistic information layers.

Finally, a last problem concerns the fact that each NLP tool programmer usually develops its own data model to represent the linguistic information, and Input/Output facilities to that data model. Since, these different data models normally represent similar information they tend to be very similar. The redefinition of such similar models represents a waste of time.

The main objective of this work is to build an NLP framework for the creation of NLP systems with the following properties:

- Avoid information losses between tools composing a system;
- Simplify new NLP tools implementation by providing general Input/Output facilities and a data model to represent linguistic information;
- Simplify the reutilisation of existing NLP tools by minimising the changes required in each individual tool;

- Maintain the data from tools aligned allowing the navigation through related data from different tools.

The proposed solution is a framework using a client-server architecture instead of a pipelined architecture. In our solution, the server acts as a repository where all NLP tools (clients) add/consult data. The server maintains cross-relations between the existing layers of data. The data is kept in the repository under a data model independent of the client tools. This data model allows the representation of a broad range of linguistic information. The tools interact with the repository through a generic remote API that permits the creation of new data and the navigation through all the existing data. Moreover, this work provides libraries implemented in several programming languages that abstract the connection and communication protocol details between NLP tools and the server, and provide several levels of functionality that simplify the creation and integration of NLP tools.

2. Solution Requirements

We defined a set of requirements for a solution that supports and simplifies the integration of independently developed NLP tools into NLP systems. These requirements were used to validate the proposed solution, and are the following:

- All information produced during the execution of a system should be available;
- Tools should only produce directly related information;
- The solution should simplify the creation of new tools, by providing: (i) an Input/Output interface, which handles the loading and saving of data used by the tool; (ii) a data model that can be used by each tool to represent linguistic information;
- The solution should minimise the number of conversion components required to build an NLP system, when integrating existing NLP tools that do not comply with the system's model;
- The provided interface should allow the navigation between information produced by different NLP tools.

To achieve the previous generic requirements we defined two groups of requirements that the solution must fulfil, namely:

- Data Model Requirements - Represents the requirements for a data model capable of representing and relating a broad range of linguistic information, which are described in Subsection 2.1.;
- System requirements - Represents requirements of the underlying system related with the interaction between the system and the NLP tools, which are described in Subsection 2.2..

2.1. Data Model Requirements

The data model main requirement is that it must be able to represent a broad range of linguistic information produced by different NLP tools. Furthermore, the data model must be extensible because it is impossible to foresee all kinds of linguistic information that may appear in the future. We begin by distinguishing two conceptually different kinds of information that the data model must represent:

- Primary data sources such as a text, or a speech signal;
- Linguistic information produced by NLP tools, over primary data sources, or previously defined linguistic information.

We also identify four types of actions that NLP tools may perform:

- Creation and edition of primary data sources, for example, the incremental creation of a new primary data source containing the phonetic transcription of the text belonging to another primary data source. This newly created data source can be the target of the linguistic information of other tools;
- Identification of linguistic elements from a primary data source, for instance, the segmentation of a sentence into words;
- Creation of relational information between linguistic elements, such as the relation between a verb and the corresponding subject.
- Assignment of characteristics to a linguistic element, or a relation, for example, the morphological features of a word;

Each NLP tool may produce several types of information at the same time. The linguistic information generated by an NLP tool is normally derived from linguistic information created by other tools. For example, a part of speech tagger will use the segments produced by a tokenizer and add morphologic information to those segments. A morphological disambiguator may use the classifications produced by several part of speech taggers to select the most appropriate classification.
The data model must be able to represent several layers of both primary data sources and linguistic information. We have defined the following requirements regarding these two types of information:

- The data model must be able to represent any kind of primary data source such as text, speech, video, or any combination of these;
- The data model must support the creation and edition of primary data sources;
- All linguistic information except primary data sources produced by an NLP tool must be associated with the same layer;
- The data model must allow the selection of linguistic information through the identification of the layer that contains it;

- Each layer is associated with the identification of the tool that produced it.

The last three requirements are necessary to simplify the identification of information inside the data model. This way all linguistic information is organised into layers.
The data model must represent the three types of linguistic information that each NLP tool can produce: (i) the identification of linguistic elements; (ii) the creation of relations between linguistic elements; (iii) and the assignment of characteristics to linguistic elements and relations.
We have defined the following requirements for the representation of those linguistic elements:

- The model must be able to represent ambiguity in the identification of linguistic elements, for example, a compound term can be segmented as only one segment containing the compound term or several segments for each word.
- The model must be able to represent trees of linguistic elements, for example, syntactic trees.
- The model must allow the creation of relations between linguistic elements from the same layer.
- It must be possible to represent classification ambiguity, which correspond to associating disjunct sets of characteristic to the same linguistic element. For example, distinct morphological features for the same word.
- The model must allow the association of characteristic to linguistic elements, or relations from other layers.

Besides the representation of linguistic information produced by each NLP tool, the data model has the following requirements concerning the relations between linguistic information from different layers:

- The model must be able to represent relations between linguistic elements from different layers. These relations represent dependencies between layers of information, and allow the navigation between layers;
- The data model must allow linguistic elements to reference data belonging to a primary data source, without having to copy its value, thus avoiding the repetition of the same data in several layers;
- The model must be able to represent data which may not exist in any primary data source, for example, the separation of contractions.

2.2. System Requirements

This subsection presents the general requirements of the system which are not related to its data model, but with the interaction between the system and the NLP tools, which are the following:

- The system must simplify the iteration of data in the *repository*, e.g. all segments from a segmentation. This is required because iteration is the most common way of interaction between an NLP tool and its data;
- Any NLP tool can select data based on a layer's identification. This way an NLP tool only needs to handle the data it requires;
- It is possible to access the data of an unfinished *Analysis*. This way an NLP tool may consume information that is being produced at the same time by another NLP tool, allowing the parallel processing of data;
- The system must make the data persistent;
- The system must be able to interact with NLP tools written in any programming language.

3. Related Work

During the development of this work we analysed several architectures whose goal was to simplify the creation or integration of NLP tools, towards their usage in NLP systems, namely: the Emdros text database system (Petersen, 2004), a text database engine for analysis, and retrieval of analysed or annotated text; the Natural Language Toolkit (Loper and Bird, 2002), a suite of libraries, and programs for symbolic, and statistical natural language processing; the Gate architecture (Bontcheva et al., 2004), a general architecture for text engineering that promotes the integration of NLP tools by composing them into a pipes and filters architecture; and the Festival speech synthesis system (Taylor et al., 1998), a general framework for building speech synthesis systems.
We also compared some work from the linguistic annotation field, whose focus is on the definition of a logical level for annotation independent of the annotations' physical format. This logical level should be able to represent the most common types of linguistic annotations to promote reuse of annotated corpora. Our data model can be seen as this logical level. In this field we compared two works: the Annotation Graphs Toolkit (AGTK) (K. Maeda and Bird, 2002) that is an implementation of the Annotation Graphs formalism (Bird and Liberman, 1999), the most cited work in this area; and the ATLAS architecture (Bird et al., 2000), a generalisation of the Annotation Graphs formalism to allow the use of multidimensional signals.
The AGTK and the ATLAS architectures do not allow the separation of information into layers. In these architectures, to avoid the loss of information, each tool has to load all previous annotations, and then save them together with its results. This strategy has several drawbacks: first, each tool must know how to manage data which may be unrelated with the tool itself. Second, each tool may have to load and parse extra data upon its initialisation and consequently save extra data when terminating. Finally, it is difficult for a tool to handle data from several tools at the same time, because it must merge the common data from the input tools. Concerning the expressiveness power of the data model used to represent linguistic information, the adoption of the Annotation Graphs model is not possible, mainly because it does not allow the representation of relational information, nor the representation of cross-relations between several data layers. The extensions performed by the ATLAS architecture provide a better representation for conceptually different linguistic phenomena, such as hierarchic trees, and ambiguous segmentations. Furthermore,

ATLAS allows the use of every type of data sources. But, even so, this model still presents the same problems as the Annotation Graphs formalism.
The Emdros framework has a representation model that is too restrictive for our objectives. For example, it restricts the media type to text. In Emdros it is not possible to properly represent some types of linguistic phenomena, such as classification ambiguity, or relations between elements.
The NLTK restricts development to the Python programming language, and relies on the Python interpreter to work. Moreover, its underlying data model is not able to fulfil all our requirements, for instance, the representation of ambiguities, such as, classification ambiguity.
The GATE framework presents the same problems as the AGTK formalism, concerning its data model. Moreover, it limits its use to the Java programming language. GATE promotes the integration of NLP tools into a pipes and filters architecture, which as we mentioned in the introduction, has some problems that led to the development of this work.
Finally, the Festival framework has a model that does not allow the fulfilment of all our requirements, namely: i) its source data must be text, ii) it cannot represent all types of ambiguities defined in the requirements, iii) it does not allow the concurrent execution of different tools.
The requirements defined in the previous section were compared against the requirements that are being defined by ISOTC37/SC4 (Terminology and other language resources) to define a standard for linguistic annotation (Ide et al., 2003; Ide and Romary, 2001). We found them to be very similar, which strengthened our conviction that any model used to represent linguistic information should follow these requirements.

4. Proposal

Our proposal consists of a client-server architecture. The clients are NLP tools while the server consists of a centralised repository of linguistic information and data sources represented under a data model. Each NLP tool can interact with the server in two ways:

- By using a remote interface independent of the tool's programming language;
- By using a module in its programming language that abstracts the communication and protocol details between the client and the server, and offers an implementation of the data model. The use of the server is simpler using the client library than using the remote API, but the use of the client library requires an implementation of the client library for each programming language used.

4.1. Data Model

The data model is able to represent and relate various types of linguistic information produced by several NLP tools. Besides representing the different linguist phenomena, the model simplifies the use of linguistic information.
The entities composing the data model, described in Fig.1 are:

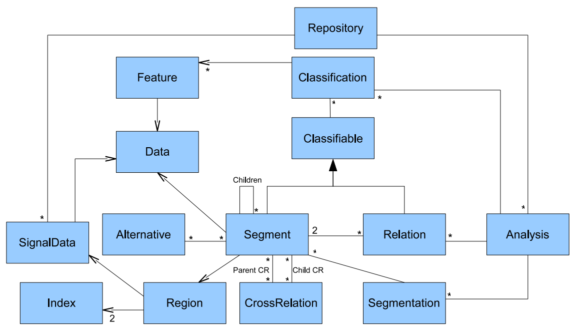


Figure 1: Data Model class diagram.

- *Repository*: a centralised linguistic repository that stores the output of several NLP tools, and organises that information into layers. Each layer is univocally identified inside the repository. There are two types of layers: *SignalData* and *Analysis*;
- *Data*: an abstraction of a data type that can be used by the *Repository*, e.g. *String*;
- *SignalData*: an abstraction of a raw data source, e.g. a text, abstracting details such as its physical location or its data type. All *SignalData* have a minimal granularity unit. In a text that unit might be a character, while in an audio file it might be the sample;
- *Index*: a point in a *SignalData* using its minimal unit;
- *Region*: defines a region in a *SignalData*, using a pair of *indexes*. The *Region* encapsulates the details about the specifics of *SignalData* and *Index* elements;
- *Analysis*: linguistic information other than *SignalData* elements produced by an NLP tool. An *Analysis* may be open or closed indicating whether the tool has already finished the addition of data into the *Repository*. An *Analysis* can only be changed if it is open. The *Analysis* is responsible for the creation of *Segmentations*, *Relations* and *Classifications*;
- *Segment*: a linguistic element, e.g., a word. A *Segment* may contain two Data elements: the original data and the derived data. The original data corresponds to a linguistic element identified in a *SignalData*, while the derived data corresponds to a possible transformation performed over the original data. A *Segment* may be ambiguous, meaning that it has a set of *Alternative* elements for the linguistic element that it identifies. It may be hierarchic, meaning that it has a *Parent Segment* and *Child Segments*. A *Segment* has a set of disjunct *Classification* elements, where each *Classification* assigns a set of characteristics to the *Segment*. It also has a set of Relations, that establish links between two Segments from the same Analysis. A *Segment* may belong to a set of *CrossRelations*, which are used to establish structural relations between *Segments* from different *Analyses*;

- *Segmentation*: a set of sequentially ordered *Segments*, e.g., the words in a sentence. The *Segmentation* is responsible for the creation of *Segments*;

- *Relation*: a link between two segments. For example, the relation between the subject and the verb in a phrase;

- *Classification*: a set of characteristics of a *Segment* or *Relation*. For instance, the morphological features of a word;

- *Alternative*: a set of *Segments* representing different alternatives for an ambiguous linguistic element;

- *Cross-Relation*: a structural relation between *Segments* of distinct *Analyses*. It is used to navigate between different layers of information.

Figure 2 shows how the derived data attribute can be used to represent the separation of a contraction. It shows a *Repository* with two layers. The first layer is a *SignalData* that contains the text *"They're waiting outside"*. The second layer is an *Analysis* with two *segmentations*. The first *Segmentation* contains a *Segment* for each token of the *SignalData* which references a *Region* in the *SignalData*. The second *Segmentation* shows the use of *derived data* to represent the separation of the contraction *"They're"*. It contains two *Segments* with *derived data* (represented in italic) one for each word composing the contraction. These segments still contain the *Region* referencing the *original data*, which allows the access to the original state of the text. To clarify the example, we have repeated the representation of the *SignalData* layer to not overlap the arrows. Note that, there is only one layer with that *SignalData*.

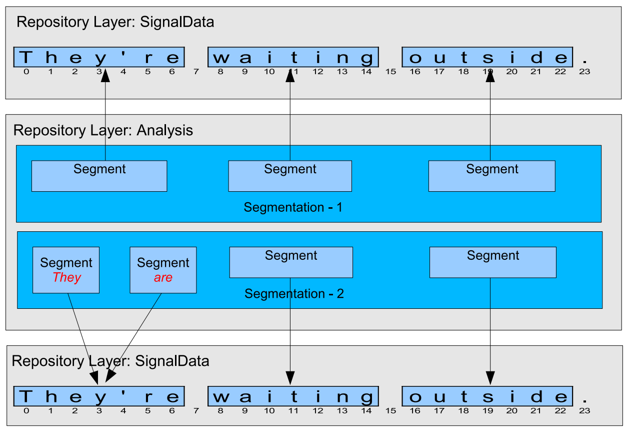


Figure 2: Use of derived data example.

Figure 3 shows the representation of an example of classification ambiguity. The figure shows a *Repository* with 3 layers. The first layer contains the original *SignalData* while the second layer contains an *Analysis* with the segmentation of the original text into words. The third layer represents the output of a part-of-speech tagger. Each *Segment* from the second layer has several *Classifications*. In this example, each *Classification* only contains one attribute-value pair: the word's part of speech. This example shows the association of *Classifications* to *Segments* from other layers.

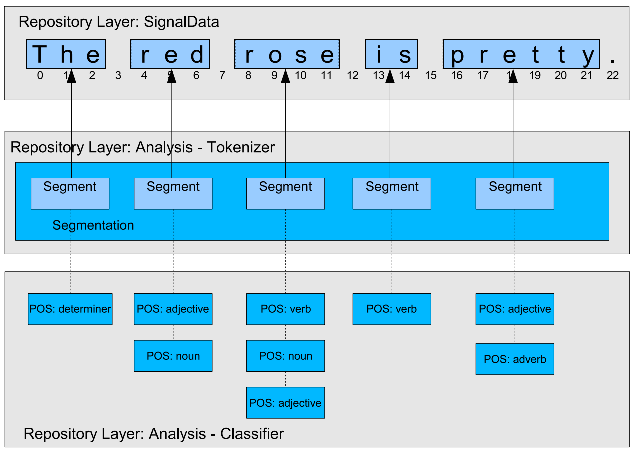


Figure 3: Classification ambiguity example.

Figure 4 shows how *CrossRelations* can be used to align different *SignalData* elements. The figure shows the translation of the sentence *"The red rose is pretty"* from English to Portuguese, *"A rosa vermelha é bonita"*. The *Repository* contains four layers. The first layer corresponds to the original *SignalData* containing the English text, where the possible *Indexes* are represented as integers under each character, and the possible *Regions* identifying the words are represented inside a box. The second layer is an *Analysis* which contains the segmentation of that text into words: it contains one *Segment* for each word, that uses a *Region* (indicated by an arrow from the *Segment* to the *Region* in the *SignalData*) to refer to the word's text. The *Segmentation* contains those *Segments* in accordance with the word's order in the text. The third layer is a *SignalData* containing the translation of the English text from the first layer, and the fourth layer is an *Analysis* produced by an English to Portuguese translator tool, which creates a *Segmentation* where each *Segment* represents a Portuguese word from the third layer. The representation of the third and fourth layers is the same as the explained for the first two layers. The alignment between the two texts is achieved by adding *CrossRelations* between *Segments* from the corresponding *Analyses*. The *CrossRelations* are represented as dotted arrows between *Segments*.

4.2. Server Architecture

The server architecture consists of a shared data style. This architecture has the advantages of allowing clients to be added without the server knowledge, and of allowing the integration of the data produced by all the tools. The linguistic information, described using the data model, is managed by the server. The server is organised in three layers: the data layer, the service layer, and the remote interface layer. Each layer can only use the adjacent layers through their interfaces. The layered approach promotes portability, and maintainability since the role of each layer is well identified, and the implementation of each layer can be changed without affecting the other layers.

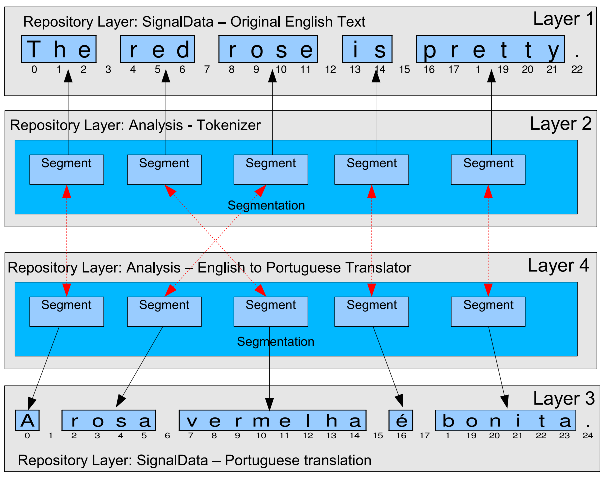

Repository Layer: SignalData – Original English Text
Layer 1
The red rose is pretty.
Repository Layer: Analysis - Tokenizer
Layer 2
Segment
Segmentation
Repository Layer: Analysis – English to Portuguese Translator
Layer 4
Segment
Segmentation
A rosa vermelha é bonita.
Layer 3
Repository Layer: SignalData – Portuguese translation

Figure 4: Data alignment example.

4.2.1. The Data Layer

The data layer contains the logic of the application, and uses the data model to represent the linguistic information. Besides representing all the linguistic information, the Data Layer is also responsible for guaranteeing the persistence of all linguistic information stored in the server.

4.2.2. The Service Layer

The service layer (Fowler, 2002) defines the server's boundary by providing a set of methods and coordinates the server's response to each method. It is used by the Remote Interface Layer, which handles the specific protocol details, and encapsulates the Data Layer. The Service Layer is responsible for hiding the details regarding the representation of the domain elements of the Data Layer. It transforms the domain elements into Data Transfer Objects (DTO) which will be passed to the client. A DTO (Fowler, 2002) is an object with no semantics, and is used to pass information between the client and the server. Each DTO can hold two kinds of information from domain objects: i) identification information used to access domain objects; ii) read-only information that may be required by the client.

The Service Layer is also responsible for providing methods that allow the creation of iteration facilities on the client side. Moreover, and since the Service Layer is a single entry point into the server,it is an ideal place to perform logging and authentication actions.

The Service Layer together with the DTOs works as a Remote Facade (Fowler, 2002) thus diminishing the number of remote calls required for certain operations.

4.2.3. The Remote Interface Layer

The remote interface layer provides the methods that are available to client tools according to a selected protocol. It communicates with the Service Layer, and is responsible for serialising the DTOs provided by the Service Layer into their external representation, which will be sent across the connection. It is also responsible for assembling the DTOs back, and pass them to the Services Layer.

4.3. Client Library Architecture

The use of the client library allows an NLP tool to abstract from details concerning the communication with the server, and the data exchange protocol. It also provides some high level interfaces that may simplify the integration of NLP tools. The client library uses a layered architecture, each layer is described in the rest of this subsection.

4.3.1. The Client Stub Layer

The client stub layer is responsible for communicating with the server under the chosen protocol, through the server's Remote Interface layer. All the other layers of the client library depend and use this layer. This way the other layers are independent of the specific communication protocol that is being used.

4.3.2. The Data Model Layer

The data model layer implements the data model. It allows NLP tools to use the data model as their object model, thus simplifying their creation. Since the concepts used by NLP tools are usually similar, by using the data model we desire to avoid the definition of an equivalent one every time a new tool is created. In addition, by using only the interfaces provided by the Data Model layer its concrete implementation can be changed without changing the tool. This way, an NLP tool can be used as a stand-alone tool or as a client tool connected to the shared repository just by changing the implementation of the Data Model Layer.

The Data Model layer elements are proxies(Gamma et al., 1995) for the elements of the Data Model in the server. The methods performed on those elements are delegated into the corresponding elements in the server.

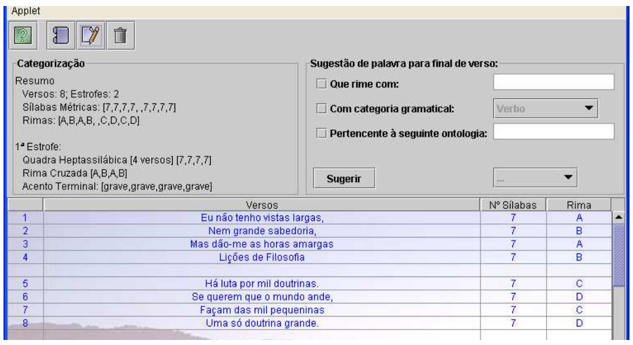
The repository can be used concurrently by several NLP tools, so it is possible that a tool consumes information that is being produced by another tool at the same time. If the consumer tool is faster than the producer tool and depletes the data that is being produced, the consumer tool may finish its processing due to a lack of data before what was expected. To avoid this situation the iterators on the client side have a blocking behaviour. The method `hasNext()` only returns false when the Analysis that contains the data that is being iterated is closed and there are no more elements to iterate. However, this policy can result in a deadlock to the consumer tool if the producer tool ends abruptly without closing its Analysis. So, we introduced a time limit for which a client method can be blocked in the method `hasNext()`.

4.3.3. The Extra Layers Layer

The extra layers layer provides extensibility to the client library. It represents new layers that can be added on top of previous ones, enabling the creation of domain specific layers, which may simplify the creation of new NLP tools. For example, a part-of-speech tagger could use a layer that provides the concepts of word, phrase and text, with methods such as `nextWord()`, and `addGender(Word w)`. The usage of Extra Layers can also provide semantic meaning to the linguistic information kept in the Repository for a given NLP system.

5. Poetry Learning Aiding System

The Poetry Learning Aiding System (Araújo, 2004)is a didactic NLP system for learning the concepts of Portuguese poetry as well as aiding the creation of poems. The system major functionalities are the characterization of poems and the suggestion of words to complete a verse. Figure 5 shows the system's user interface. The user can add a poem contained in a text file to be characterized, or ask for a suggestion to end the last verse of the current poem.


Applet

Categorização

Resumo
Versos: 8; Estrofes: 2
Sílabas Métricas: [7,7,7,7, ,7,7,7,7]
Rimas: [A,B,A,B, ,C,D,C,D]

1ª Estrofe:
Quadra Heptassilábica [4 versos] [7,7,7,7]
Rima Cruzada [A,B,A,B]
Acento Terminal: [grave,grave,grave,grave]

Sugestão de palavra para final de verso:
☐ Que rime com:
☐ Com categoria gramatical: Verbo
☐ Pertencente à seguinte ontologia:
Sugerir

	Versos	Nº Sílabas	Rima
1	Eu não tenho vistas largas,	7	A
2	Nem grande sabedoria,	7	B
3	Mas dão-me as horas amargas	7	A
4	Lições de Filosofia	7	B
5	Há luta por mil doutrinas.	7	C
6	Se querem que o mundo ande,	7	D
7	Façam das mil pequeninas	7	C
8	Uma só doutrina grande.	7	D

Figure 5: Poet User Interface example.

The system is composed by several tools, some of them were already developed in our group, while others where developed specifically for this system. The initial input of the system is a text file containing the source poem where verses are separated by blank lines. The system contains the following tools:

- The poem structure identifier - This tool places the original poem in the repository and creates two analyses. One containing the segmentation of the poem into verses and the other containing the separation of the poem into lines. These two analyses are aligned using cross-relations (each line is cross-related with the corresponding verse);

- A word tokenizer and part of speech tagger - This tool segments the poem into words (these new segments are aligned with the sentence segments) and assigns each word's possible part of speech tags as several classifications;

- Syllable segmentation and phonetic transcriber - This tool segments each word into its grammatical syllables (kept aligned with the corresponding words) and adds a classification for each syllable containing its phonetic transcription;

- Poem characterization tool - This tool classifies the poem based on the information produced by the previous tools. It uses for instance, the number of syllables of each sentence, the number of sentences of each verse and the the type of rhyme of each verse;

- Word suggester - This tool suggests a word to end the last line of the last verse. It uses a n-gram model that based on the pos tags of the last words of the last line suggests the part of speech required. Then based on the phonetic termination of the last words from the other lines in the verse suggests a word to end the verse. This suggestion may be restrained to a certain pos tag or to a certain phonetic termination if required.

Figure 6 shows an example of the repository during the execution of the system before the execution of the poem characterization. At this stage all information required for the next two tools is cross-related in the repository.

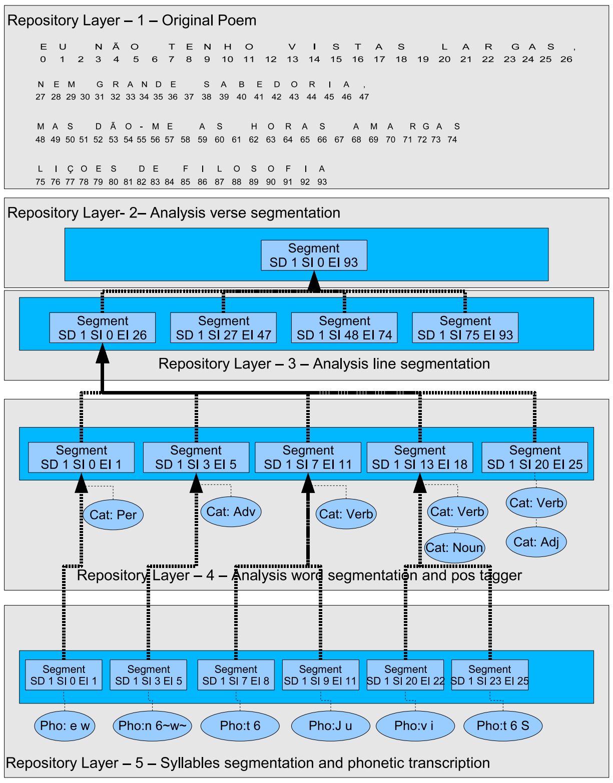


Figure 6: Poet Repository state example.

This system is the first being created using the repository and it allowing us to test the feasibility of the framework. The existing tools (pos-tagger and the phonetic transcriber) where integrated using converters from their existing formats into the repository data model format. The new tools were implemented using the data model as their object model, avoiding some issues which are usually the developer's responsibility, as the definition of an object model, and the definition of an IO interface. Moreover, having the different levels of information cross-related the creation of the poem characterization tool and the word suggester was simplified, since instead of having several input files from each tool and having to merge all information, each tool accesses one particular level and then navigates through the information using the existing cross-relations.

6. Results and Future work

We implemented our framework in *Java* (around 57 classes) using the XML-RPC protocol provided by the APACHE XML-RPC package. This protocol was chosen because of its simplicity and because it does not impose any restrictions on the programming language used by client

tools. Each layer of the server defines a set of interfaces that are used by the other layers. We have implemented a set of specialised classes in the Data Layer to handle text input signals (*TextSignalData*,*StringData*,*TextIndex*). The Service Layer is implemented by two classes: one to implement the methods from its interface, and the other responsible for assembling DTOs from domain objects. The Remote Interface layer is implemented by a class registered in the XML-RPC server as a handler class.
We implemented a client library (around 40 *Java* classes) using the XML-RPC protocol. The Extra Layers layer contains four specific classes to handle text, providing more meaningful methods to NLP tools developers.
Moreover, we defined a standoff annotation format in *XML* defined by a *DTD* for our data model. For now this format is being used to keep the data persistently, but we intend to use it to allow the standalone annotation of corpora.
This implementation fulfils all the requirements defined, allowing the representation of a broad range of linguistic phenomena, and complies with the requirements being defined by the ISO committee.

6.1. Future Work

We plan to implement more types of primary data sources, e.g. audio files, and thus enable the usage of other NLP systems, like natural language generation.
The development of some work in our laboratory, such as, character identification in stories, anaphora resolution, semantic analysis, were suffering from the problems identified in this work. We expect that the use of our framework, that allows them to access all the information produced by NLP tools, and to navigate through related information using cross-relations, simplifies their creation.
Another problem regarding the integration of NLP tools consists in the tags used by each tool to classify linguistic elements. Even if the data structure between two tools is the same, if they use different tag sets to classify the linguistic elements, they will not be able to communicate. This problem was addressed in (Matos, 2005). Since, this framework established a single entry point for the assignment of classifications, a conversion between tag sets could be performed to guarantee that inside the repository all tags were represented in the same way. Each tool would indicate which tag set it requires, and receive the information with the proper tags.
We wish to promote this framework as an annotation framework, and to do so, a graphical interface has to be developed to allow the edition of its data. Moreover, some converter modules have to be defined to allow the usage of data annotated in other formalisms, for instance, the *Annotation Graphs* formalism, in which, large corpora have already been annotated, and are publicly available. This particular usage of the framework will allow the annotation of the corpora existing in our laboratory.
As for issues that require future research, we notice that this framework does not answer two important questions regarding the integration of NLP tools. First, how does each tool know what data it must fetch from the repository. For this problem we pretend to integrate the repository with a workflow mechanism and a browser. This will enable the creation of NLP systems, by simply selecting from a browser, which tools the system should have. The browser will deal with the selection of the data for each tool. Another question is how does an NLP tool interpret the data kept in the repository. Our data model is capable of representing all linguistic information. However, the same linguistic information may be represented in different ways according to its usage. For example, if the phonetic transcription of a text is going to be the target of several NLP tools it should be represented as a new data source. On the contrary, the transcription of each word can be represented as a word's attribute if it is not going to be heavily used. In this work we assume that each tool knows the exact representation of the data. This approach might be too restrictive in terms of extensibility. Some research work should be done in this field, namely in the definition of a meta language, that allows each tool to define its data pre-requirements and pos-requirements, and a way to match this information automatically.

Acknowledgements

This paper was partially supported by project POSI/PLP/41319/2001.

7. References

P. Araújo. 2004. Classificação de poemas e sugestão das palavras finais dos versos. Master's thesis, IST - UTL.

S. Bird and M. Liberman. 1999. A formal framework for linguistic annotation. Technical Report MS-CIS-99-01, Philadelphia, Pennsylvania.

S. Bird, D. Day, J. Garofolo, J. Henderson, C. Laprun, and M. Liberman. 2000. Atlas: A flexible and extensible architecture for linguistic annotation.

K. Bontcheva, V. Tablan, D. Maynard, and H. Cunningham. 2004. Evolving GATE to Meet New Challenges in Language Engineering. *Natural Language Eng.*, 10.

M. Fowler. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns*. Addison-Wesley Professional, January.

N. Ide and L. Romary. 2001. Standards for language resources. In *Proceedings of the IRCS Workshop on Linguistic Databases*, pages 141–149, University of Pennsylvania, Philadelphia, 11-13.

N. Ide, L. Romary, and E. de la. 2003. International standard for a linguistic annotation framework.

H. Lee K. Maeda, X. Ma and S. Bird, 2002. *The Annotation Graphs Toolkit (Version 1.0): Application Developer's Manual*. Linguistic Data Consortium, University of Pennsylvania, January.

E. Loper and S. Bird. 2002. Nltk: The natural language toolkit. *CoRR*, cs.CL/0205028.

D. Matos. 2005. *Construção de Sistemas de Geração Automática de Língua Natural*. Ph.D. thesis, IST - UTL, July.

U. Petersen. 2004. Emdros - a text database engine for analyzed or annotated text. In *Colling*.

P. Taylor, A. Black, and R. Caley. 1998. The architecture of the the festival speech synthesis system.