Towards Reusable NLP Components

Amalia Todirascu, Eric Kow, Laurent Romary

Towards Reusable NLP Components

Amalia Todirascu [*], **Eric Kow** [*], **Laurent Romary** [*],

[*]Equipe Langue et Dialogue, INRIA
LORIA, Campus scientifique, BP 239, 54506 Vandoeuvre Cedex
{todirasc, kow, romary}@loria.fr

Abstract

We propose a methodology for transforming an NLP module into a reusable component, to be integrated it into a distributed and open architecture. We illustrate the methodology by showing the adaptations needed tfor an LTAG parser to be transformed into a bundle of parsing and lexical services.

1. Introduction

The availability of various NLP tools and linguistic resources for several languages opens the possibility of reusing and sharing them. Unfortunately, most of the tools provide low portability (in particular due to application-specific resource representation). Building an NLP tool or adapting it for other languages requires considerable effort in creating or updating linguistic resources (often incomplete) as well as in integrating into new applications.

This paper proposes a methodology to transform an elementary NLP module into a reusable component, integrated into an open and distributed NLP architecture. The methodology requires input/output standardization, the choice of a standard format for parameter resources and the definition of a standardized communication protocol between various components. We experimented this methodology with a Lexicalized Tree Adjoining Grammars (LTAG) parser, illustrating the problems arising in a real and somehow complex situation.

The underlying objective of this work was originally to be able to integrate such a parsing component into various larger NLP environments corresponding to various degrees of complexity from a language engineering point of view: a man-machine dialogue system in the context of the IST/MIAMM project[1] (a dialogue environment providing speech, graphical and haptic interface to a musical database), an information extraction environment (a message filtering application on computer security in collaboration with the EADS/MSI company), an on-line network of NLP services based on GRID techniques [2].

2. General background and objectives

2.1. Existing initiatives towards the integration of NLP components in unified architectures

A lot of research efforts have concentrated on the issue of building reusable NLP components within specific integrating frameworks. Several challenges were identified (**?**) when transforming existing NLP modules into reusable components: software challenges (defining APIs for each type module), semantic challenges (the output of various modules are not always semantically consistent), 'political' challenges (the algorithm and resources are not always publicly available for the research community).

Several projects propose environements for building NLP complex applications, based on existing modules. GATE is one of the most widely known, providing an environement for integrating several NLP tools, from various platforms, using readily NLP modules (a POS tagger, a parser, a discourse expert, etc.) (**?**). GATE supports the reuse of resources (as CELEX (**?**) and WordNet (**?**)) and algorithms, and provides technical support for the I/O interface, graphical interface and client-server structure for NLP tools. The I/O interface conforms with a standard TIPSTER annotation module. It provides such interesting features as JAPE (regular expressions over annotations) and GUK (enhanced unicode support). Despite its widespreadness, it requires huge amount of computer resources (memory, disk space) on a single computer environment, it is not user friendly and it imposes a pipe-line architecture. The SVENSK project aims also at reusing resources and algorithms for Swedish (**?**), using GATE. From another perspective, SiSSA (**?**) is an infrastructure for prototyping and validating NLP application architectures. It supports different languages and platforms, it uses the XML recommendation for data interchange, it allows the reuse of various processors and it is flexible as it only requires that each new module be registered at a central SiSSA manager. Each module implements a CORBA interface to communicate with the others.

All these projects require an implementation of I/O interface API (which is application-dependent) for each module. We do not intend to propose another NLP architecture, but we propose a standard methodology for transforming NLP tools into reusable components, based on existing standards for protocol and I/O interface. Our methodology is based on simple XML-like format for exchanging data, free of security constraints (as in CORBA). The architecture is completely descentralized and flexible.

2.2. Towards the definition of a parsing service

LTAG parsers are good examples of low reusability of resources: existing parsers use a variety of resource formats (XTAG grammar (**?**), Feature Tree Adjoining Grammars (FTAG (**?**), SGML (**?**)). Complete resources are avail-

[1]see http://www.loria.fr/projets/MIAMM

[2]see http://www.loria.fr/projets/Guirlande

able for English (lexicon and grammar), but they are not really available for French or for other languages. Resources and parsers are intimately related so that it is far from to, say, use en XTAG grammer as parameter of another parsing module. Still, some experiments have been conducted in providing grammar servers on-line (**?**), but with no intent to connect such ”services” to parsing components proper. In this context, the standardization of TAG resource formats was a necessary step and has resulted in the TAGML proposal (**?**), an XML application for representing elementary TAG trees and forests [3]. To illustrate the methodology, we adapted the Lopez parser (**?**) to use TAGML-based linguistic resources and to provide XML-based output. The component is incorporated into a distributed architecture, where it is combined with an independant resource server and user interface. Those two components, as we will see in this paper, have been designed to be fully independant from the specificities of the parser. In particular, sharing resource servers avoids redundancy, duplication or creation of new resources. One of our aims here was to make sure that the parsing results could be used both as an independant resource, but also to annotate the primary linguistic content provided as an input to the parser. This induced some constraints on the definition of the transmission protocol which should had the specific feature of keeping track of resource reference within the architecture.

3. Standardization

One of the crucial points to achieve an easy deployment and re-use of NLP modules is to make sure that there exists, for a given type or class of NLP components, the right standards for representing the various linguistic knowledge structures that will be used as input, output or parameter for this module. Unfortunately, even if there has been numerous attempts to define common formats for such object as lexica (e.g. Genelex), basic annotations (Eagles guidelines for part of speech tagging (**?**)), or multilevel annotation (CES, Mate), none of these initiatives have led to internationally approved standards, essentially because of the lack of a wide recognition independant from any funded project or industrial pressure group. As a direct consequence, most of the existing NLP architectures or environment have developed their proprietary description formalisms for specifying the data transiting between internal components, as we have observed in the comparative table (see figure **??**).

In this context, the creation of the new committee TC3/SC4 within ISO provides the first real international framework
General perspective — syntax/semantics
• Wide agreement on W3C as a stable technical background — XML
• Necessity to provide ways to specify one’s format while ensuring interoperability principles with similar data that would be provided or consumed by other components (kind-of semantics)

[3]A revised version of TAGML is under discussion to provide more coverage of implementers’ needs (representation of derivation trees, beeter representation of feature structures (closer compatibility with the TEI FS chapter) and should be published in the near future

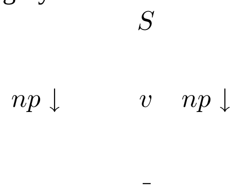
Go deeper into this (cf.(**?**)) by identifying, in a given format the underlying information organization (metamodel) and what can be seen as a parameterization of such structures (data-categories).
Cf. Part of speech tagging:
While there can be a general agreement on the basic mechanisms needed to represent data tagged for POS (embbeding of word and multiword units, alternatives to cope with ambiguities etc.), each tagger will implement its own tagset corresponding both to the language that it is dealing with and the granularity of description that one aims at. In the context of international standardization, there is thus a need to provide reference sets of data categories that a given application will use to define its own subset.
Still, it can be useful to suggest a reference syntax (or a family thereof) for such specific layers as POS tagging that may make things even easier for someone who does not want to implement his own dialect.

To illustrate how we standardized the existing linguistic TAG resources, we present some properties of TAGML, chosen as a resource format, and the input/output format.

3.1. Application: TAGML resources

TAG (**?**) lexicons associate with lexical entries all the syntactic trees which represent all the possible combinations of the lexical entry with other words. These trees (elementary trees) are combined during the parsing process by two operations: substitution and adjunction. To avoid redundancy, several trees are associated to word lemmas.

Elementary trees contains several node types: standard nodes (no leaf nodes), anchor nodes (a leaf node containing the word associated to the tree), foot nodes (a leaf node indicating a possible adjunction) and substitution nodes (a leaf node indicating a substitution). Trees containing adjunction nodes will be adjuncted to the nodes of the same category as the foot node.



The substitutions are represented by $\downarrow$ while the adjunction are represented as *. The operations are identified by their adress in the tree. The nodes are counted from left to right from 1 to n. ‘

Example. Adress 2.1 means that the operation is done at the first son of the V node.

The results of the parsing process are a set of derived trees (the syntactic structure) and a set of derivation trees (the history of substitutions and adjunctions used to build the derived trees).

The substitutions are represented in trees as continuous lines, while adjunctions are represented as dotted line.

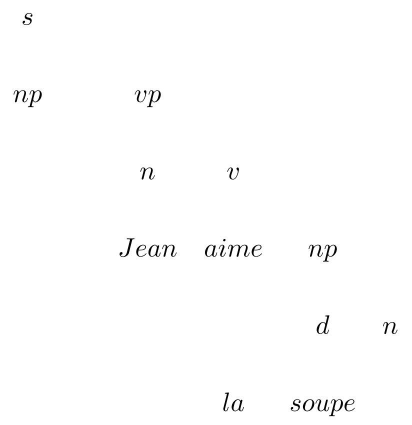
3.1.1. Input/Output Standard

XML (W3 Consortium (**?**)) is used by many NLP applications, for its adequacy for representing annotated text and for structuring data, to its extensibility and to its transparency. As a natural choice, we use for our component XML for input/output.

	GATE	**SiSSA**	**Soapical (us)**
goal	1. provides I/O, display, client-server support 2. reusability of components	1. infrastructure for prototyping, editing, validating NLP application architectures	foo
features	1. plug-in modularity of text processing components 2. handles the data storage and module loading 3. CREOLE - readily available modules 4. niceities: JAPE (regular expressions over annotations), GUK (enhanced Unicode support)	1. FIST: grammar metaformalism 2. Grammar Repository implementation 3. SiSSA Manager for registering new modules; gui for selecting how modules connect together	1.
tech	1. JDBC for accessing language resources; 2. JavaBeans standards to integrate LR -specific viewers/editors and to integrate modules into CREOLE	1. all processors have to implement an CORBA interface for SiSSA objects 2. XML for data 3. the Manager and a number of modules (processors) 4. use RDF to store info about modules	1.

Figure 1: Comparison between GATE, SiSSA and MATE

The input of the parser is a phrase in natural language and a set of resources (lexicon, grammar). The format of the lexicon and grammar will be explained in the next subsection.



The output of the LTAG parser is a set of derived trees (syntactic structures) and a set of derivation trees (a dependency structure). This reflects all the possible syntactic and dependency structures associated to the existing input phrase.

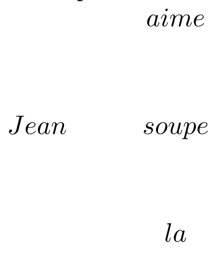
The derived trees are represented as XML elements as:

```
<n address="0" cat="s" lex="" type="0">
<n address="1" cat="np" lex="" type="1">
<n address="1" cat="n" lex="Jean" type="2" /n>
</n>
```

```
<n address="2" cat="vp" lex="" type="1">
<n address="2.1" cat="v" lex="" type="1">
<n address="2.1.1" cat="v" lex="aime" type="2"/>
</n>
<n address="2.2" cat="np" lex="" type="1">
<n address="1" cat="n" lex="soupe" type="1">
<n address="1" cat="d" lex="la" type="2">
</n>
</n>
</n>
</n>
```

The **n** element contains several attributes: the lexical category **cat**, the address were the operation has been done **address**, the node type **type** (standard, anchor, foot, substitution), the word **lex**.

Example. The derivation tree for the input phrase *Jean aime la soupe* is



and it is represented in XML as:

```
<subs string="aime">
<subs string="Jean"/>
<subs string="soupe">
<adj string="la"/>
</subs>
</subs>
```

Two types of elements are used: **subs** for tracing substitution operations, and **adj** for adjunction operations.

3.1.2. The Resources

Resource format is an important issue when reusing data and algorithms. We propose a simple, standard format. We chose to represent our linguistic resources in a standard format: TAGML (Tree Adjoining Grammar Markup Language). TAGML is also a XML-based format: it is easy to use by linguists and it structures the linguistic resources (a set of lexical entries, related to the associated elementary trees). The availability of TAGML resources is provided by tools (**?**) translating existing grammars (XTAG grammar, FTAG) or generating them from a meta-grammar.

TAGML proposes a representation of all the elements of the TAG lexicon: lexical entries, lemmas and elementary trees. TAGML structures the TAG resources on three levels: lexical entries, lemmas and tree families. Lexical entries are associated to lemmas and have some specific morpological features as number, tense, mode.

Example of a lexical entry in a TAGML lexicon

```
<morph lex="aimera">
       <lemmaref cat="v" name="*AIMER*">
           <fs>
               <f name="num"><val>sing</val></f>
               <f name="mode"><val>ind</val></f>
               <f name="tense"><val>fut</val></f>
           </fs>
       </lemmaref>
</morph>
```

lemmaref contains the pointer to the lemma, and the **fs** element contains the morphological feature structure.

Lemmas contain pointers to tree families and some constraints imposed on the feature structures associated to elementary trees. **anchor** keeps the pointer to the family tree and the equations impose some constraints on various nodes of the tree (the attribute **restr** is plus for the node labeled np at the 0 address).

```
<lemma cat="v" name="*AIMER*">
       <anchor tree_id="family[@name=tn1]">
           <equation node_id="np_0" type="top">
               <fs>
                   <f name="restr"><val>+</val></f>
               </fs>
           </equation>
       </anchor>
   </lemma>
```

Tree families contain several elementary trees grouped by their syntactic properties. For example, the following family describes the properties of complex noun phrases, containing relative clauses. A **tree** is a set of **nodes**. Each **node** has a category, a type (standard, anchor, foot, subst, lex), a property **adj**, allowing or not adjunctions and a set of constraints on each node (**narg** elements).

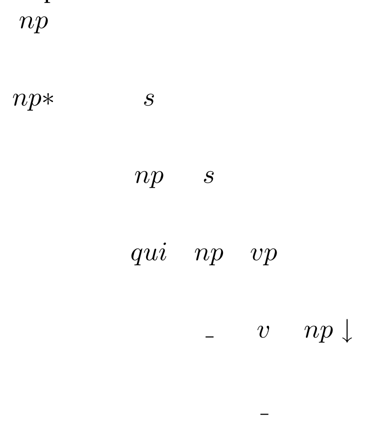
```
<family name="tn1">
       <tree name="r0tn1">
           <node cat="np" type="std">
               <node cat="np" type="foot">
                   <narg type="top">
                       <fs>
                           <f id="X0" name="num" />
                           <f id="X1" name="pers" />
                           <f id="X7" name="restr" />
                       </fs>
                   </narg>
               </node>
               <node cat="s" adj="no" type="std">
                   <node cat="np" type="std">
                       <node lex="qui" type="lex" />
                   </node>
                   <node cat="s" type="std">
                       <narg type="bot">
                           <fs>
                               <f name="inv"><minus /></f>
                               <f id="X2" name="mode" />
                           </fs>
                       </narg>
                       <node cat="np" id="np_0" adj="no" type="std">
                           <narg type="top">
                               <fs>
                                   <f id="X7" name="restr" />

                      </fs>
                           </narg>
                           <node type="lex" />
                       </node>
                       <node cat="vp" type="std">
                           <narg type="bot">
                               <fs>
                                   <f id="X3" name="mode" />
                                   <f id="X5" name="num" />
                                   <f id="X6" name="pers" />
                               </fs>
                           </narg>
                           <narg type="top">
                               <fs>
                                   <f id="X2" name="mode" />
                                   <f id="X0" name="num" />
                                   <f id="X1" name="pers" />
                               </fs>
                           </narg>
                           <node cat="v" type="anchor">
                               <narg type="bot">
                                   <fs>
                                       <f id="X3" name="mode" />
                                       <f id="X5" name="num" />
                                       <f id="X6" name="pers" />
                                   </fs>
                               </narg>
                           </node>
                           <node cat="np" id="np_1" type="subst" />
                       </node>
                   </node>
               </node>
```

```
            </node>
        </tree>
        <tree name="r1tn1">

        </tree>
</family>
```

is represented as:



3.2. SOAP protocol

As our system uses the parser and the resource accessing module installed as services on various sites, we need a communication protocol between various components. SOAP is a XML-based standard (**?**) for exchanging messages between components of a distributed application, and we chose it because of its simplicity and flexibility. SOAP provides a homegenoeous way of handling all data: messages, input/output, resources.

There are two ways to look at SOAP. The first is on a higher level, as a format for messages between agents in any distributed system. A SOAP message consists of an envelop, which wraps a header and a body. This header is where standard SOAP metadata, such as the actors required to understand a message, go, and the body, the actual data we are interested in. While SOAP does also offer some of its own metadata, exceptions (faults), and some default primitives (ints and strings, for example), it is actually a more useful understanding of the protocol to pretend that in its entirety, SOAP is only composed of those three tags: envelop, header, and body. Because it specificies exactly so little, we are free to design the contents of the body and add things to the header at will. For instance, our SOAP bodies would merely be in TAGML (see figure **??**). This is what we mean by flexibility.

On a second, lower level, we can think of what people actually do with SOAP, or rather, how it is used as a communication protocol. One path of very little resistance is to bind these SOAP enveloppes to HTTP headers and send the resulting message along its way. In fact, this is what most default implementations of SOAP do; they are little programs (CGI scripts, servlets) that can be attached to webservers. Any software that uses such a SOAP implementation thus only worries about producing and consuming data, leaving it to the SOAP implementation to wrap such data into SOAP envelops, or unwrap such SOAP envelops and turn it back into useful data. Some sophisticated implementations, like Apache SOAP even go a step further, and provide a Remote Procedural Call mechanism on top of this, such that function calls are translated into SOAP messages, unwrapped by the server, then called accordingly, with the resulting data then being itself wrapped and sent back to the client. Once attached to a such a SOAP implementation, a client then concerns itself with function calls, and the server to the provision of these functions. This is what we mean by simplicity.

We feel that SOAP will bring us to our goal of having a widely adopted methodology towards reusability, and we feel this way not because SOAP is any more technologically interesting than other object-sharing mechanisms, but because it is easy. Rather than dictate plausibily correct specifications for inserting an NLP module into specific architecture, we instead propose a protocol that specifies just enough: data and metadata. It is lightweight, built upon pre-existing standards, tied to familiar transport mechanisms, and has thus been widely implemented. Even the Mozilla web browser has a SOAP implementation. In short, we find SOAP to be useful because its standardness and familiarity provides a path towards reusability, but at a mental cost low enough to compete with the ad hoc approach favoured by the computational linguist of today.

4. System architecture

To experiment this methodology, we split the parser into various modules: the module implementing the connected-routes-based algorithm (**?**), the visual workbench and the resources. We integrate all these modules into a distributed, flexible architecture. Each module contains a XML-based communication level for data exchanges, via the SOAP protocol. We used for tests small resources for French (338 lexical words, 50 trees) and English (279 words, 421 trees). Some extra modules, for resource aquisition (lexicon, annotated corpora) might be added into the architecture.

Users enter its texts via a graphical interface. The input texts are sent to the parser, sentence by sentence. The parser creates an instance for each sentence and it sends a request to the resource server to access LTAG resources. The parser will be a client of the Lexicon server. The lexical entries together with the elementary trees are sent back to the parser which continues the process. It sends the resulting forest of derivation and derived trees back to the user interface. The user interface lets the user browse the forest tree by tree. The resource server contains a maintenance module updating the existing resources with the parser output (validated by the user via the interface).

4.1. The services

The initial parser has been splitted into several services: the core of the parser acting as a central service installed on a server, the resource access module (loading the grammar and the lexicon) and the graphical user interface.

4.1.1. The parser

The parser implements a connected-routes-based algorithm (**?**) for TAG grammars. It builds syntactic structures from elementary trees, combining them by substitution and adjunction. The output of this parser is a forest of derivation and derived trees. Optimisations (unification of feature structures during parsing, other algorithms) are still under development. Its input is a phrase in natural language. It asks the lexicon server for the trees associated with each

Figure 2: a simplified example of a SOAP message

```
<?xml version='1.0' encoding='UTF-8'?>

<SOAP-ENV:Envelope ...>
<SOAP-ENV:Header/>
<SOAP-ENV:Body>

<tag><family name="tmpfam_Jean"><tree name="np">
           <node cat="np" id="np_" type="anchor"/>
 </tree></family></tag>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

input word of the phrase. When the trees are returned to the parser core, the parsing process tries to build derivation and derived trees associated to the phrase and even if the process fails to produce a syntactic structure for all the input phrase, it returns a set of partial derived and derivation trees.

4.1.2. The grammar

The lexicon server is structured into three levels: morphological level (containing flexed words and pointers to lemmas), the lemma level (containing the lemmas, the features associated with lemmas and pointers to trees) and the tree level (containing families of trees). If a word is not found in the lexicon, it returns an null answer.

It returns a set of trees associated to the input phrase. For some applications (Vulcain, MIAMM), we need to select only domain-specific lexicons. We intend to use several lexical services, using existing lexical services.

4.2. The metaservices

We also deploy in this architecture a set of reusable diagnostic modules. The first of these is the SOAPMeter, which is to be inserted between two SOAP nodes, such as a parser and its lexicon service. Once in place, the SOAPMeter analyses the communication between the two nodes, and displays them as a series of individually timestamped SOAP messages. This way, the implementor knows instantly if instead of an actual problem with the modules themselves, there is a lower level problem, say, that the reason the parser is not doing anything useful is that it is not receiving any requests from the client. Once freed from these concerns, the implementor then proceeds to a higher level, seeing a series of requests and responses, rather than losing his place in the flow of HTTP headers and XML text.

To further illuminate these requests and responses, the SOAPMeter acts as a SOAP client for our PRISM [4] service, which receives XML text and returns a graphical representation of that text. XML messages can be challenging to read because so much of the message is dedicated to describe a document's structure, after wading through which, one does not attain a solid grasp on its contents or their relationship to each other. With a graphical representation, we no longer describe the structure, but *show* it,and furthermore, show it in a way that is relevant to the specific data at hand. Syntatic trees, for example, can be actual trees instead of block upon block of open/close XML tags. Likewise, if we were dealing instead with a logical description (such as Discourse Representation Structures DRSs) it would be possible to visualise it as something more appropriate (for DRSs, using the classical "box" representation).

In PRISM, we acheive this parameterisability through the use of XML stylesheets (XSL) that describe the transformation of XML to graphics, which is to say that, changing the nature of a SOAP diagnosis is adding or changing a stylesheet rather than modifying any of the diagnostic software. As for the SOAPMeter, this leaves the implementor with a single button-click to visualise any SOAP messages which might be particularly interesting or difficult.

5. Applications

The system was developped in order to integrate the parser into complex applications: the MIAMM project (Multimedia Information Access using Multiple Modalities), developping a multi-modal interface (combining haptic, text, speech, graphics) to search songs in a database and the Vulcain project, dedicated to message filtering about computer security. The applications have various degrees of complexity and the resources are very different: Vulcain is an information-extraction system using local grammars and domain-specific lexicons, but MIAMM uses resources handling dialogue and reference resolution.

6. Conclusion

The paper illustrates a methodology for transforming a NLP module into a reusable, parametrizable component for a particular case: a LTAG parser. The methodology requires input/output and resource format standardization (XML-compatible) and the definition of a communication protocol (SOAP). The methodology will be validated by transforming other NLP modules into reusable components, accessing resources available on-line. The SOAP protocol will be extended with more complex functions (handling XML references, implementing XML query for updating resources, extending the mechanisms for handling errors). The parser component will be integrated into an open architecture handling human-machine dialogues.

[4]Parameretisably Readable Interpretation of SOAP Messages

Figure 3: System architecture