MAE BASED TOOL FOR SEMANTIC ANNOTATION OF A SOURCE CODE

Nebojša Sarvan,
Nina Manojlović,
Milica Nikolić,
Goran Savić,
Milan Segedinac*

Faculty of Technical Sciences,
University of Novi Sad,
Novi Sad, Serbia

Abstract:
The motivation for this research came with the need for a tool that can be easily used for annotating the source code and which would provide the most suitable output for latter processing and analysis. The output should represent all of the pieces of annotated code followed by annotation given, and additional features. These features refer both to the text of the code and annotation such as a field that expands the meaning of annotation and begin and end positions in a document of the text and annotation. For the purpose of annotating source code we have used an existing tool MAE. In order to facilitate retrieving of an output this paper proposes a solution to transforming MAE output and mapping it to another more suitable form.

Keywords:
text mining; text annotation; annotating tool.

1. INTRODUCTION

Nowadays, when a huge amount of data is available all over the Internet, it is very important to collect and properly analyze them as they can be very valuable for various purposes. In order to process textual data correctly, one could rely on variety of tools for text mining. For better understanding and providing computers to deal with data as good as humans, general semantic of the text should be recognized. Semantic web is the most popular approach when it comes to this. Words and phrases are represented by ontologies, which help computers determine text meaning. In order to put individual words into context, all the words should be given an annotation [1, 2]. Text annotation is a process of adding an additional meaning to the text, by marking it, highlighting or commenting. It turns the content into a better manageable data source [3]. From this point, text annotations can be referred to metadata as they provide information about a text without fundamentally altering its original form [4].

The problem with semantic annotations is that these annotations are not universal. Therefore, domain specific knowledge is used for semantic annotation and this domain specific information is provided by ontologies [3, 5]. In the case considered, semantic annotation is strictly limited to the field of IT, more specifically to annotating source code.

Correspondence:
Milan Segedinac

e-mail:
milan.segedinac@gmail.com

DOI: 10.15308/Sinteza-2018-31-37

Dealing with a source code annotation is a way more distinct from just annotating regular text. It introduces new kind of complexity, where more structured forms are expected for defining and describing concepts those annotations are related to.

This paper proposes a specific tool for annotating source code relying upon the existing annotation tool MAE and explains in details how to use MAE in combination with our specific tool in order to get the appropriate results. Its main goal is to describe the process of mapping the output from this tool to a convenient form for later processing in some of IDE's (*Integrated Development Environments*). Although the paper is more concentrated on combination of these two tools, our software can also be used independently for other kinds of purposes.

In addition to this introduction, the paper consists of four sections and a reference list. Section I presents related work through analysis of some of techniques used for annotation and explanation how our solution should overcome spotted disadvantages, followed by the detail coverage of MAE annotation tool. Section II fully describes our work. All of modifications done to adjust the MAE tool for the purpose of our research are presented and our software for working with MAE is fully specified. Concrete case study is presented and discussed in Section III. Finally, Section IV gives an overview of the paper and some possible directions of how this system can be upgraded in the future.

2. RELATED WORK

When firstly introduced to our task, the main goal was to find an existing tool for source code annotation, rather than writing it from the scratch. However, among plenty annotation tools it was not that easy finding a convenient one. Beside all the good features that tools we came across during our research provide, they were also lots of disadvantages that would require additional tasks in order to use them, which did not seem as the best solution.

ELAN [6, 7] is the software designed for creating, editing or searching annotations for audio or video files that is a very powerful tool when it comes to annotating. However, based on the fact that is principally designed for multimedia and not text annotations, we assumed that modifying its source code would take more time than implementing our own annotation tool from beginning.

When it comes to text mining, more precisely text extraction, preprocessing and analysis, GATE [8, 9] has very good characteristics and provides well performing results. More importantly, GATE is capable of automatic text annotation, relying on existing ontologies. On the other side, due to its very complex structure and extra task on making an ontology that reflects our data, GATE did not represent the best possible tool for our task.

Eclipse plugin [10] is an additional feature provided by *Eclipse* Integrated Development Environment (IDE) that deals with source code annotations. Even though it fulfills our needs in annotating source code, utilization of this plugin is narrowed just to one specific programming environment and it also incorporates all attached annotations directly within source code, which makes the original source code messy and harder to understand.

What we needed was a simple tool that allows manual text annotation, whether it is free plain text or a source code. Besides, just selecting specific lines or sections, freedom in annotating the code with annotations that fit the task most properly, and setting our own annotating structure was what really matters. We have found MAE [11, 12] as the most suitable tool for our work for two reasons. Its ability to provide for annotating source code represents the biggest MAE advantage. Another reason for choosing MAE is that it is an independent tool, which holds and handles files for annotation by itself, rather than incorporating it in original source code which could change its original structure and make it more inconspicuous.

MAE annotation tool

MAE (*Multi-document Annotation Environment*) is an annotation tool for natural language text annotation. MAE is written in Java, thus Java version 8 or later is required for MAE to run. It is available in executable binary (jar) file or as a release package (zip).

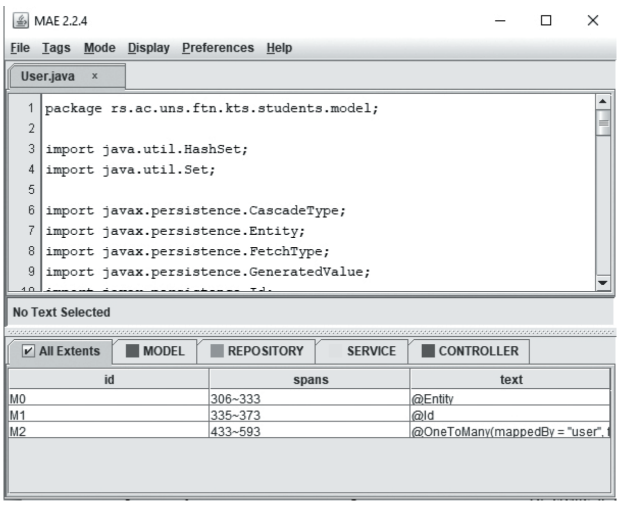
Defining an annotation task is first thing to do in order to use MAE tool properly. The task is represented by DTD (*Document Type Definition*) which defines the structure and the legal elements and attributes of the data. By so, defining the task name, the tag names, and the tag attributes are the necessary steps in process of completing an annotation task. The task name is defined with the *!ENTITY* tag, followed by specified word *name* and actual name of the task you want to be created. In annotated output files this is reflected as the name of the root tag element. Tag elements are defined by

!ELEMENT and specify the names of the tags being in an annotation task, and their attributes. Although two types of tags can be specified – extent tags and link tags, we are going to concentrate just on explanation of the extent tag, since it is most important for our work. Extent tags should be unique, and they are used to label spans of text in the document. *#PCDATA* follows the name of an extent tag and indicates that it is going to be associated with some span of text in the document. Each extent tag is assigned a color to visualize tag instance over the document that is being annotated. Attributes (defined by the *!ATTLIST*) keep the information associated with each tag, and expand their meaning. Some attributes are pre-defined by MAE – extent tags will always have attributes like *id*, *spans*, and *text*, even if they are not determined in the DTD. In order to define attribute, first thing to do is to specify an extent tag name which attribute is going to refer to. Later, attribute name and type should be given. Those types will be mentioned later in the paper. Firstly, we will go over the details of the pre-defined attributes in MAE.

While creating the tags, MAE will automatically assign an *id* attribute to every each of them. This attribute provides all the tags to be uniquely identified. As mentioned earlier in the chapter, all extent tags have an attribute called *spans*, which denotes the positions of first and last character of annotated area within the document. However, it is possible for an extent tag to be "non-consuming". By default, MAE does not allow an existence of tags like this, but by defining an extent tag's *spans* attribute as *#IMPLIED*, MAE will allow that tag to be non-consuming.

In regard to attribute types, MAE supports four types, from which only one is relevant for our task. All of the attributes are of type *CDATA* which means that they have a free text value. Attribute can be set to mandatory or optional, which is accomplished by assigning them keywords *#REQUIRED* and *#IMPLIED* respectively. All of the required attributes should be filled. MAE allows setting default values for any attribute by placing the desired value at the end of the attribute definition.

Once a valid task definition is created, it could be loaded into this tool. MAE will generate tables corresponding to tags defined in the DTD in the bottom half of the interface (Fig. 1). As mentioned earlier, to each extent tag type, different color will be assigned.



```java
package rs.ac.uns.ftn.kts.students.model;

import java.util.HashSet;
import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
```

id	spans	text
M0	306~333	@Entity
M1	335~373	@Id
M2	433~593	@OneToMany(mappedBy = "user",

Fig. 1. MAE's Graphical User Interface

MAE can be used for annotating plain text files as well as an existing XML annotation task, so both of those kinds of files could be loaded. An existing XML file can be opened only when it matches the name of task definition that is currently loaded (the root node of the XML). Otherwise MAE will open the XML as a plain text. The document will be showed up at the top half of the interface. Once the file for annotation is loaded, annotation process can begin. By selecting the desired annotation from the menu, tag instance is created and corresponding row in a table in the bottom is immediately populated. MAE will automatically generate predefined attributes, and fill in *id*, *spans* and *text* fields. When a new tag is created, MAE will assign any default values for its attributes, if specified in DTD.

3. SYSTEM FOR SOURCE CODE ANNOTATION

In previous section we have introduced an existing annotation tool – MAE and fully described it in its original form. This part will also describe MAE, but now from the point of our, modified form. All of modifications will be mentioned and explained. Also, reasons for those modifications will be given. Later, we'll present how the output of the modified MAE tool can be integrated and used for purpose of source code annotation.

MAE modifications

As already mentioned, main goal of this research is annotation of the source code for educational purposes. MAE does the job when it comes to annotating source code, but for our specific task it failed in giving good results. Rather than doing all the work from the beginning, we have decided to use the existing solution, and upgrade it. First of all, it's important to emphasize that we didn't modified the source code on its own, but only made changes of their original DTD file.

As described in section II, MAE has quite complex structure. In order to simplify it, we left out multiple parts of its DTD file, which are surplus and irrelevant for our particular problem. These parts mainly refer to link tags and their attributes. What have left of original DTD was later modified in a way to support structure of files needed to be annotated. The files are the part of student project with unique structure that will be described in details in the next section.

Fig. 2 represents DTD file when modified.

```
<!ENTITY name "NounVerbTask">


<!-- #PCDATA makes a extent tag-->
<!ELEMENT MODEL ( #PCDATA ) >
<!-- this can be non-consuming-->
<!ATTLIST MODEL spans #IMPLIED >
<!-- fixed value set with a default value-->
<!ATTLIST MODEL comment CDATA "" >
<!ELEMENT REPOSITORY ( #PCDATA ) >
<!ATTLIST REPOSITORY comment CDATA "" >
<!ELEMENT SERVICE ( #PCDATA ) >
<!ATTLIST SERVICE comment CDATA "" >
<!ELEMENT CONTROLLER ( #PCDATA ) >
<!ATTLIST CONTROLLER comment CDATA "" >
```

Fig. 2. Appearance of the modified .dtd file

DTD consists of four main elements – MODEL, REPOSITORY, SERVICE and CONTROLLER. Each element has the same attribute list, as follows: *id*, *spans*, *text*, and *comment*. First three attributes are already described in previous section. The last-mentioned attribute is added to expand the meaning of each element, more precisely to give the element semantic connotation. This attribute is represented by free text input field, and is left empty for an annotator to fill it. There is no specific standard for that, but we established the convention in order to have unique structured data that can be easily processed. Element titles were not chosen randomly, but in a way to follow the problem definition. Reasons for declaring those titles and also *comment* field structure will be given in section IV. Now, let's turn to specification of our specific software and how it deals with the output that modified MAE tool provides.

Our software

This software is, above all, made as an upgrade of MAE tool. However, it can be used independently for similar kinds of problems. The main goal is to do mapping of features given by MAE into specific form suitable for later analysis. This chapter provides information about software implementation, both as a guide of how to use it.

Software is written in Java programming language, so it is basically Java application. Input is the output of the MAE tool, more specifically, the XML files generated as a result of annotating process. XML contains all kind of knowledge of a file that was annotated using MAE. In our case, annotated file consists of source code, instead of plain text. *TEXT* tag contains this kind of information. It is followed by tag TAGS which represents all of user defined annotations. As mentioned in Chapter II (A), these tags consist of attributes needed for describing particular annotation. One of the main reasons for creating this tool is the fact that the most important attribute is not in proper form for further analysis. Specifically, attribute *span* contains range between first and last character position of annotated area in the file that has been annotated. The way we need it to be is some kind of format that provides information about line in which annotated area starts and ends, both as columns with same meaning. So, our first task was to do the mapping from one to another appropriated form. This particular task required parsing XML files first, so we could gain originally annotated source code as well as access *span* attribute at all. One may not have all of the original files on its file system, so one way of accessing it would be to extract it from TEXT tag and save it as another temporary file. After having those values, one should call inbuilt *LineFromChar* function, as follows:

```
    LineFromChar(File tempFile, int startCh-
ar, int EndChar)
```

where *tempFile* refers to location of temporary file, *startChar* represent position of first written character while *endChar* is position of last written character in annotated area. Return value of this function is a String that contains all the knowledge about lines and columns in which annotated text starts and ends. Due to numerous annotations with same title that can be found in multiple different files, output of our tool is represented as a collection of sections, looking as follows:

```
    Section(String fileName, String content,
String sectionType, int lineStart, int col-
Start, int colEnd, int lineEnd, String com-
ment)
```

where *fileName* is a name of original source code file which contains this particular annotation, content is a piece of source code covered by annotation, *lineStart*, *lineEnd*, *colStart* and *colEnd* are respectively line in which annotated text starts, line in which annotated text end, column in which annotated text starts and column in which annotated text ends. *SectionType* as well as *comment* parameter are strictly tied to our case study, thus will be explained in details in Chapter IV.

This collection is mapped to JSON format, as it is the most suitable detected form for latter tasks. The whole process is shown in Fig. 3.

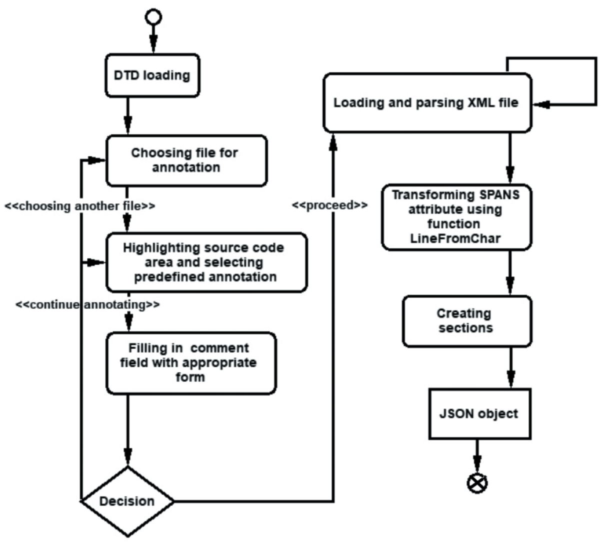
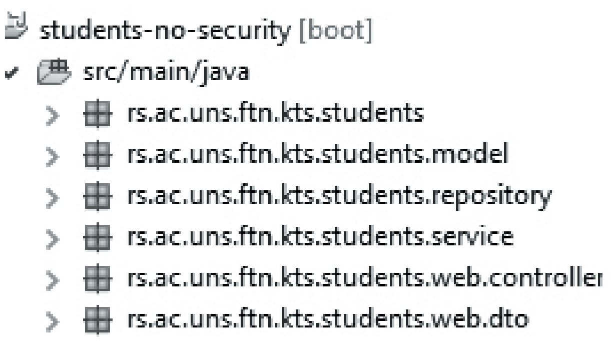

DTD loading
Choosing file for annotation
<<choosing another file>>
Highlighting source code area and selecting predefined annotation
<<continue annotating>>
Filling in comment field with appropriate form
Decision
<<proceed>>
Loading and parsing XML file
Transforming SPANS attribute using function LineFromChar
Creating sections
JSON object

Fig. 3 . Annotation process represented by Activity **Diagram**

4. CASE STUDY

All of the work is done as a part of major project which purpose is to determine student's knowledge in IT domain, based on eye movements. For this task Java Spring project is used as a case study, so model for predefined annotations and annotations themselves are specified for Java source code. Structure of project that is used is shown in Fig. 4.



```
students-no-security [boot]
  src/main/java
    > rs.ac.uns.ftn.kts.students
    > rs.ac.uns.ftn.kts.students.model
    > rs.ac.uns.ftn.kts.students.repository
    > rs.ac.uns.ftn.kts.students.service
    > rs.ac.uns.ftn.kts.students.web.controller
    > rs.ac.uns.ftn.kts.students.web.dto
```

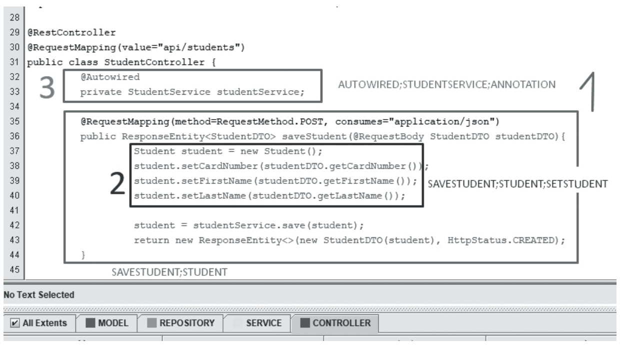
Fig. 4 . Structure of a project used for case study

Based on project structure one can easily notice few obvious sections which we declared as predefined annotations. Followed by names of the packages, those annotations are titled as: *MODEL, REPOSITORY, SERVICE* and *COTROLLER*. Some of the classes does not carry any additional meaning, thus they are not relevant for this specific task and were not annotated. Each annotation has the same attributes which were explained in details in previous chapters. The most specific of them all is *comment* attribute, which is tightly connected to this particular case study. Depending on type of section, particularly if it is annotation in code, or part of a method, comment field can have three forms. Comment is a plain text field which should be written in one of the following formats:

- `METHOD_NAME;ENTITY` , if section is referring to the whole method. In form above, *method_name* represents name of the method which is being annotated, and *entity* stands for entity that the previous mentioned method is related to, mostly return value of specific method.
- `METHOD_NAME;ENTITY;SUBMETHOD_DESCRIPTION` , if section covers a piece of source code that is nested inside of a more complex method. *Submethod_description* is a name (similar to the method name) that shortly describes logic behind highlighted part of method.
- `ANNOTATION_NAME;ENTITY;"annotation"` , if section is referring to an existing annotation. *Annotation_name* is the name of annotation (in our case Java annotation) and *entity* represents entity which has been annotated by previously mentioned Java annotation. The third part of the comment is a constant value "*annotation*" which indicates that a section is placed over Java annotation.

The form of this field is not standardized, but should follow mentioned convention.

Fig. 5 shows an example of all of the three forms of comment field.



```
@RestController
@RequestMapping(value="api/students")
public class StudentController {
    @Autowired
    private StudentService studentService;

    @RequestMapping(method=RequestMethod.POST, consumes="application/json")
    public ResponseEntity<StudentDTO> saveStudent(@RequestBody StudentDTO studentDTO){
        Student student = new Student();
        student.setCardNumber(studentDTO.getCardNumber());
        student.setFirstName(studentDTO.getFirstName());
        student.setLastName(studentDTO.getLastName());

        student = studentService.save(student);
        return new ResponseEntity<>(new StudentDTO(student), HttpStatus.CREATED);
    }
```

Fig. 5. Possible forms of comment field

Part of the image marked with 1, is related to the first case where annotation represents the whole method. Blue box, numbered with 2, gives an overview of a piece of source code which is taken as a section, but is located within some other, more complex method. Number 3 is Java annotation section.

As mentioned before, annotated source code is sent to further processing, and its output is represented by JSON file format. The snippet of the file is shown in Fig. 6.

```
{       "fileName": "ExamService",
        "content": "public void remove(Long id) {   examRepository.delete(id);  }",
        "sectionType": "SERVICE",
        "lineNumStart": 28,
        "colNumStart": 1,
        "lineNumEnd": 30,
        "colNumEnd": 2,
        "comment": "remove;Exam"
}, {
        "fileName": "Student",
        "content": "@Entity public class Student {",
        "sectionType": "MODEL",
        "lineNumStart": 15,
        "colNumStart": 0,
        "lineNumEnd": 16,
        "colNumEnd": 22,
        "comment": "Entity;Student;annotation"
},{
        "fileName": "Student",
        "content": "@Id  @GeneratedValue  private Long id;",
        "sectionType": "MODEL",
        "lineNumStart": 17,
        "colNumStart": 1,
        "lineNumEnd": 19,
        "colNumEnd": 17,
        "comment": "Id;id;annotation"
},
```

Fig. 6. Snippet of a JSON output file

The output represents every piece of annotated source code followed by an annotation and some additional features. It keeps information about:

- *fileName* – name of the file that contains this piece of annotated code
- *content* – text of the source code that is annotated
- *sectionType* – name of the predefined annotation, assigned to the selected piece of code
- *lineNumStart* – line number where annotation begins
- *colNumStart* - column number where annotation begins
- *lineNumEnd* – line number where annotation ends
- *colNumEnd* - column number where annotation ends
- *comment* – field that expands annotation meaning

5. CONCLUSION

The main goal was to develop a software tool that can easily be used for annotating source code. Following the results of analysis of appropriate existing annotation tools, we have chosen MAE tool for the basic annotation task. Once the source code has been annotated, results are held in XML files which all integrated represents an input to our tool. This tool parses XML files and applies a set of functions on original format in order to transform it in a more convenient format for further analysis. The result is represented in JSON file format.

Even though our DTD file contains predefined annotations referring mainly to the structure of Java Spring project, the tool is not strictly restricted to just one programming language.

Regarding to all the work done, we have implemented only the parts tightly connected to our specific task.

The current version of the tool has one practical disadvantage which is that all the functionalities are provided only as a pure source code. That fact pretty much reduces the group of potential users and indicates that extra effort should be put to make executable file followed by graphic user interface (GUI) as to simplify work with this tool.

Another important issue is that our work was used in combination with MAE tool, which has its own specific output format. In order to expand usage of our software and make it compatible with other tools, writing a more generic parser which could be able to parse different types of files, should be considered.

Finally, beside functionalities described in this paper, there are still lots of possibilities for enhancement in the field of text mining, extraction and interpretation of the data. All of these are directions for future work.

REFERENCES

[1] Bo Pang and Lilian Lee, Opinion Mining and Sentiment Analysis, Foundations and Trends in Information Retrievel, Vol. 2, No. 1–2, pp. 1-135, 2008.

[2] Vishal Gupta and Gurpreet S. Lehal, A Survey of Text Mining Techniques and Application, Journal of Emerging Technologies in Web Intelligence, Vol. 1, No. 1, pp. 60-76, 2009.

[3] Simone Teufel, Jean Carletta and Marc Moens, An AnnotationSscheme for Discourse-level Argumentation in Research Articles, Proceedings of the Ninth Conference on European Chapter of the Association for Computational Linguistics. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 110-117, 1999.

[4] Oscar Corcho, Ontology Based Document Annotation: Trends and Open Research Problems, International Journal of Metadata, Semantics and Ontologies, Vol. 1, No. 1, pp. 47-57

[5] Xian Wu, Lei Zhang and Yong Yu, Exploring Social Annotations for the Semantic Web, Scotland , 2006, ISBN:1-59593-323-9

[6] Maddalena Tacchetti, User Guide for ELAN Linguistic Annotator, 2017, Available online: http://www.mpi.nl/corpus/html/elan_ug/index.html

[7] Peter Wittenburg, Hennie Brugman, Albert Russel, Alex Klassmann and Han Sloetjes, ELAN: a Professional Framework for Multimodality Research, In Proceedings of the 5th International Conference on Language Resources and Evaluation, pp. 1556-1559, 2006.

[8] René Witte, Qiangquiang Li, Yonggang Zhang and Juergen Rilling, Text Mining and Software Engineering: An Integrated Source Code and Document Analysis Approach, IET Software, Vol. 2, pp. 3-16, 2008.

[9] Ian H. Witten, Katherine J. Don, Michael Dewsnip and Valentin Tablan, Text mining in a digital library, Vol. 4, No. 1, pp. 56-59, 2004.

[10] Eclipse official documentation, Resource Marker, Available online: https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_markers.htm

[11] Amber Stubbs, MAE and MAI: Lightweight Annotation and Adjudication Tools, Proceedengs of the 5th Linguistic Annotation workshop, pp. 129-133, 2011, ISBN: 978-1-932432-93-0

[12] Kyeongmin Rim, MAE2: Portable Annotation Tool for General Natural Language Use, In Proceedings of the 12th Joint ACL-ISO Workshop on Interoperable Semantic Annotation, Slovenia, 2016.