EDITS - Edit Distance Textual Entailment Suite Version 1.0 - User Manual

Bernardo Magnini[1], Milen Kouleykov[1], and Elena Cabrio[12]

[1] FBK-Irst, Trento, Italy
[2] University of Trento, Italy
{magnini,kouylekov,cabrio}@fbk.eu

May 22, 2009

Abstract. EDITS is a software package aimed at recognizing entailment relations between two portions of text, termed as *text* **T** and *hypothesis* **H**. The system is based on edit distance algorithms, and computes the **T**-**H** distance as the cost of the edit operations (i.e. insertion, deletion and substitution) that are necessary to transform **T** into **H**. EDITS is based on three main modules: an edit distance algorithm, a cost scheme for the three edit operations, a set of rules representing transformations between portion of **T** and **H**. Each module can be configured by the user through a configuration file. The system can work at different levels of complexity, depending on the linguistic analysis carried out over **T** and **H**. Both linguistic processors and semantic resources that are available can easily be used within EDITS, resulting in a flexible, modular and extensible approach to Textual Entailment.
This document introduces the main functionalities of the system and provides an in depth description of how EDITS can be used, how available tools and resources can be integrated, and how the system can be extended.
EDITS has been developed by the HLT research unit at FBK-irst (http://hlt.fbk.eu/) and can be downloaded as open source software from the EDITS web site (http://edits.fbk.eu/).

Acknowledgments. Work on EDITS has been partially supported by the EU-FP6 QALL-ME project (http://qallme.fbk.eu/). We thank Matteo Negri and Yashar Mehdad, who contributed to test the system and to revise this document.

Table of Contents

1 Introduction

EDITS is a software package aimed at recognizing entailment relations between two portions of text (see [1] for an introduction to textual entailment). The package assumes the task definition as proposed by the RTE (Recognizing Textual Entailment) evaluation campaigns[3] [2]. According to the two-way classification task at RTE, EDITS takes as input a Text (**T**) and an Hypothesis (**H**) and outputs one of two judgements, either *YES* or *NO*, supported by a confidence score.

EDITS implements a distance based approach for recognizing textual entailment, which assumes that the distance between **T** and **H** is a characteristic that separates the positive **T**-**H** pairs, for which the entailment relation holds, from the negative pairs, for which the entailment relation does not hold. More specifically, EDITS is based on edit distance algorithms, and computes the **T**-**H** distance as the cost of the edit operations (i.e. insertion, deletion and substitution) that are necessary to transform **T** into **H**.

The edit distance approach implemented in EDITS is based on three core modules:

- an *edit distance algorithm*, which calculates the minimal set of edit operations that transform **T** into **H**;
- a *cost scheme*, which defines the cost associated to each edit operations;
- optional sets of *rules*, both *entailment rules* and *contradiction rules*, providing specific knowledge (e.g. lexical, syntactic, semantic) about allowed transformations between portion of **T** and **H**.

Each module can be configured by the user through a configuration file (i.e. the EDITS configuration file - ECF).

EDITS can work at different levels of complexity, depending on the linguistic analysis carried out over **T** and **H**. An internal representation format, called **ETAF** (EDITS Text Annotation Format) is defined such that both linguistic processors and semantic resources that are available can easily be used within EDITS, resulting in a flexible, modular and extensible approach to Textual Entailment.

Given a certain configuration of its three basic components, EDITS can be trained over a specific RTE dataset in order to optimize its performance. In the training phase EDITS produces a *distance model* for the dataset, which includes a distance threshold S, $0 < S < K$, that best separates the positive and negative examples in the training data. During the test phase EDITS applies the threshold S, so that pairs resulting in a distance below S are classified as *YES*, while pairs above S are classified as *NO*. Given the edit distance $ED(T, H)$ for a **T**-**H** pair, a normalized entailment score is finally calculated by EDITS using the following formula:

$$entailment(T, H) = \frac{ED(T, H)}{(ED(T, \_) + ED(\_, H))} \quad (1)$$

where $ED(T, H)$ is the function that calculates the edit distance between **T** and **H** and $(ED(T, \_) + ED(\_, H))$ is the distance equivalent to the cost of inserting the entire text of **H** and deleting the entire text of **T**. The entailment score has a range from 0 (when **T** is identical to **H**), to 1 (when **T** is completely different form **H**).

A detailed description of the edit distance approach implemented in EDITS can be found in [4].

2 Getting Started with EDITS

This Section provides a quick introduction to the main functionalities of the EDITS package. More details can be found in the rest of the document. After having installed EDITS (Section 2.1), there are four main steps required in order to use EDITS:

1. Configure EDITS main parameters (Section 2.2)
2. Prepare the RTE datasets (both training and test), providing the linguistic annotations required by the selected distance algorithm (Section 2.3)
3. Run EDITS over a training dataset (Section 2.4)
4. Run EDITS over a test dataset (Section 2.5)

[3] http://pascallin.ecs.soton.ac.uk/Challenges/RTE/

2.1 Install EDITS

EDITS can be downloaded from *http://edits.fbk.eu*. Updated versions will be periodically released and uploaded at the same web page. EDITS runs on Unix-based Operating Systems and has been tested on MAC OSX, Linux and Sun Solaris. The system requires SUN Java, version 6 [4].

In order to have EDITS installed it is enough to unzip the package in any directory of your machine. The basic organization of files and directories in EDITS is shown in Figure 1.

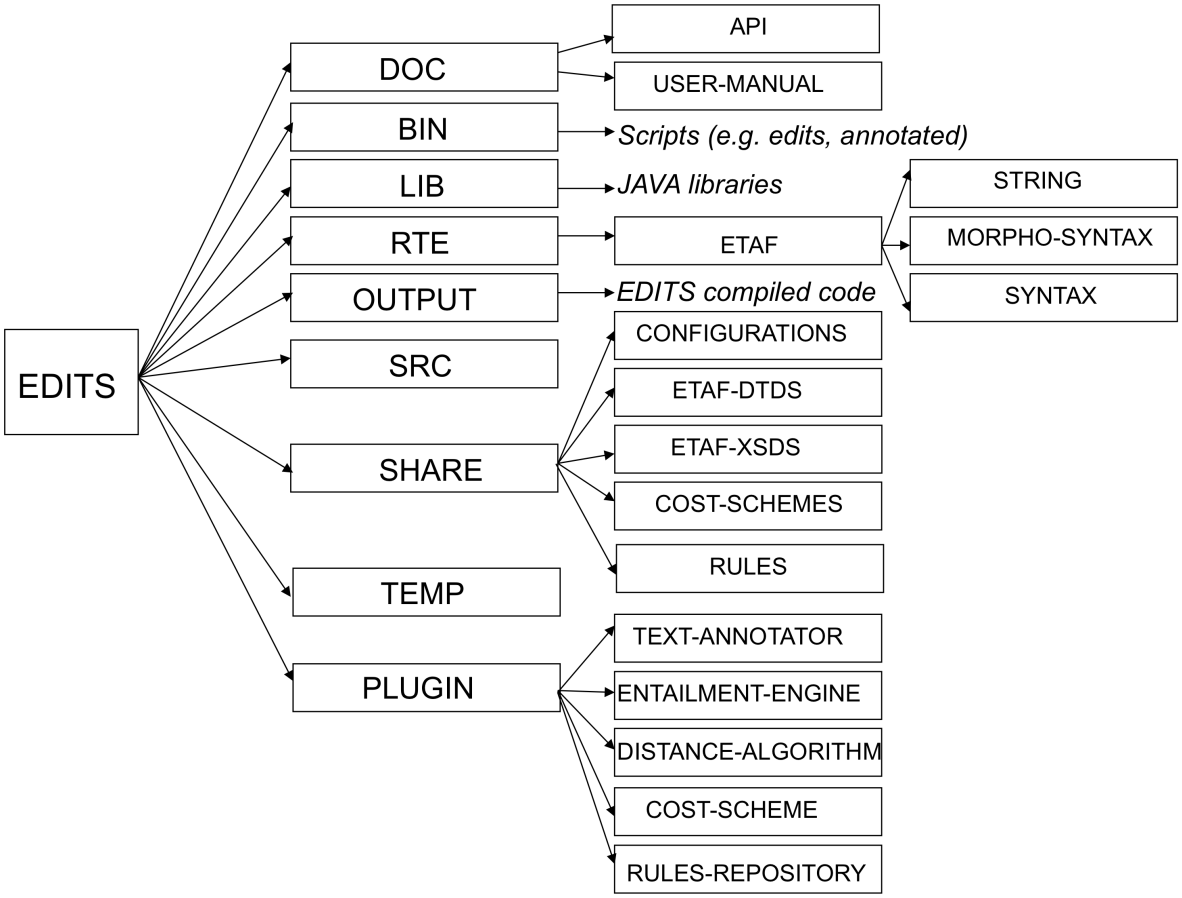

Fig. 1. Organization of EDITS files and directories

2.2 Configure EDITS

The simplest way for getting started with EDITS is to choose a pre-defined EDITS Configuration File (ECF) among those available in the `edits/share/configurations` directory. As an example, the following is the configuration file *conf2.xml*:

```
<conf>

   <module type="entailment-engine">
      <!-- Active modules in the configuration -->
          <mlink idref="tree-edit-distance"/>
          <mlink idref="xml-cost-scheme"/>
          <mlink idref="entailment-rules-wordnet"/>
   </module>
```

[4] The system can be used with SUN Java 5 by adding the Java Architecture for XML Binding (JAXB) 2.0.5 jars (https://jaxb.dev.java.net/) to the lib directory.

```
   <module id="tree-edit-distance"
          type="distance-algorithm"
          className="eu.fbk.hlt.edits.distance.algorithms.TreeEditDistance"/>

   <module id="xml-cost-scheme"
          type="cost-scheme"
          className="eu.fbk.hlt.edits.distance.cost.scheme.XmlCostScheme">
          <attribute name="scheme-file">${EDITS_PATH}/share/cost-schemes/simple-rules-scheme.xml
          </attribute>
   </module>

   <module id="entailment-rules-wordnet"
          type="rules-repository"
          className="eu.fbk.hlt.edits.rules.DefaultRulesRepository">
          <option name="entailment-rules">${EDITS_PATH}/share/rules/entailment-rules-wordnet.xml
          </option>
   </module>

</conf>
```

In the upper part of the file (entailment engine), the modules that EDITS will actually use in the current configuration are declared using a reference to their IDs: the algorithm (`mlink idref="tree-edit-distance"`), the cost scheme (`mlink idref="xml-cost-scheme"`) and the rules (`mlink idref= "entailment-rules-wordnet"`).

In the rest of the ECF, there is a list of definitions of the modules that the system can use. Each module is referred by its name (`id`) and its function (`type`). The attribute of the modules can contain information about resources used by the modules, or information about their behaviour. For instance, the attribute "scheme file" of the "xml-cost-scheme" module specifies the path to the directory containing the cost scheme. It is possible to list different modules of the same type (e.g. different distance algorithms), and to choose the one you want to use changing the `idref` in the upper part of the configuration file.

N.B. It is necessary to list at least one distance algorithm and one cost scheme.
More details on ECF can be found in Section 7.

Concerning distance algorithms, EDITS provides, as pre-defined, both *string*, *token* and *tree* edit distance algorithms (for more details, see Section 4). Pre-defined cost schemes (i.e. the module that determines the cost of the three edit operations) are available in the directory: `edits/share/cost-schemes/`

As an example of cost scheme, the one reported below (i.e. the *simple-scheme.xml* used in conf1.xml) allows to set the cost of insertion to 10, the cost of deletion to 10, and the cost of substitution to 0 when the two tokens to be substituted (referred by the functions (**attribute "token"** A) and (**attribute "token"** B) respectively) are the same, and to 20 when the two tokens are different. For more details on cost schemes see Section 5.

```
<schemes>
<scheme>
    <!-- A simple scheme for cost function calculation -->
    <!-- Valid constant types are : number boolean string and file -->
    <constant name="INSERTION" type="number" value="10"/>
    <constant name="DELETION" type="number" value="10"/>
    <constant name="SUBSTITUTION" type="number" value="20"/>
    <insertion name="insertion">
        <cost>INSERTION</cost>
    </insertion>
    <deletion name="deletion">
        <cost>DELETION</cost>
    </deletion>
    <substitution name="equal">
        <condition>(equals (attribute "token" A) (attribute "token" B))</condition>
        <cost>0</cost>
    </substitution>
    <substitution name="not-equal">
```

```
        <condition>(not (equals (attribute "token" A) (attribute "token" B)))</condition>
        <cost>SUBSTITUTION</cost>
    </substitution>
</scheme>
</schemes>
```

Concerning entailment and contradiction rules, each rule is defined with a left-hand context, the text **T**, and a right-hand context, the hypothesis **H**, and a probability that the rule preserves either the entailment or the contradiction between **T** and **H**. As an example, the following two rules state that *invent* entails *pioneer* with probability 1, and that *beautiful* contradicts *ugly* with probability 1.

```
<rule name="1">
   <t><string>invent</string></t>
   <h><string>pioneer</string></h>
   <probability>1.0</probability>
</rule>

<rule name="a">
    <t><string>beautiful</string></t>
    <h><string>ugly</string></h>
    <probability>1.0</probability>
</rule>
```

EDITS provides a pre-defined file of entailment rules extracted from WordNet (i.e. the synonyms and the hypernyms of the words contained in the RTE datasets). This file can be found in the directory `edits/share/rules`.

2.3 Preparing the RTE datasets

Both the input and the output of EDITS are RTE datasets, i.e. set of **T**-**H** pairs expressed in the same XML format as defined in the RTE evaluation campaign. An example of a RTE pair (from the RTE2 development set) is the following:

```
<pair id="61" entailment="YES" task="IE">
  <t>Although they were born on different planets, Oscar-winning actor Nicolas Cage's new son
     and Superman have something in common - both were named Kal-el.</t>
  <h>Nicolas Cage's son is called Kal-el.</h>
</pair>
```

The edit distance algorithms used by EDITS often require that the RTE dataset is processed by linguistic tools. For instance, since a tree edit distance algorithm works over the syntactic structure of **T** and **H**, both training and test data need to be processed by a parser. EDITS provides an internal format (called ETAF, see Section 3) for representing linguistic annotations. The format is enough flexible and extensible to accommodate most of the morpho-syntactic and syntactic information useful for Textual Entailment, and the user should provide converters from the format of the linguistic tools he/she wants to the ETAF format.

In order to facilitate its use, EDITS provides external interfaces [5] for a couple of existing tools: TextPro [7], a suite of text processors for Italian and English (tokenizer, lemmatizer, named entities recognizer), and the Stanford Parser [3], a dependency parser for English. Such interfaces are provided separately from the system, as plugins, and they do not contain the annotation tools, which have to be obtained separately and installed according to the readme instructions of the interfaces. In order to be used, external interfaces have to be installed and then the annotation process is called over a certain RTE dataset through the command **bin/annotate**. Here below the command usage with the required parameters.

```
Usage: annotate [-option(=value)?]* input output

input           The file/files containing the RTE pairs
```

[5] Interfaces are downloadable from http://edits.fbk.eu

```
output          The annotated RTE file/files

-annotator=name    The name of the linguistic tool used for annotation: textpro, stanford-parser  [optional]
-words          Separates a text into words using a tokenizer
-language=lang  The language of the entailment corpus: English (default) Italian [optional]
-conf=path      Configuration file path [optional]
-verbose=level  The verbose level of system messages: 0 (default) 1 2 [optional]
-overwrite      Overwrites the output files if they already exist [optional]
```

As an example, the following command annotates the development set of RTE2 with TextPro, and puts the output of the process (i.e. the annotated file) in the directory `rte/etaf/morpho-syntax`.

```
bin/annotate -annotator=textpro -overwrite rte/etaf/string/RTE2_dev.xml RTE2_dev_textpro.xml
```

Obviously, the same annotation process using TextPro should be performed also on the test set.

2.4 Train EDITS

Once EDITS has been configured and the RTE datasets have been annotated, the system can be run over a training dataset. The command **bin/edits** can be used both for training (with the **-train** option) and testing (with the **-test** option). The *input* is a training_set file or directory containing an RTE corpus already annotated with the ETAF format; the *output*, the model_path, is the path to a file specified by the user where EDITS will store the obtained distance model.

In addition the **bin/edits** command allows for a number of options, reported below.

```
Usage: edits -train [-option(=value)?]* training_set model_path

-conf=path      Configuration file path [optional]
-algorithm=name    Algorithm to use. Possible values: string (String Edit Distance -default) token
                (Token Edit Distance), tree (Tree Edit Distance) [optional]
-task=value        Calculate the threshold per task of the entailment pair: false (default) true
                [optional]
-length=value   Calculate the threshold per length of the entailment pair: false (default) true
                [optional]
-scheme=path    Path to a cost scheme file [optional]
-rules=path     Path to an entailment rules file [optional]
-simple=path    Path to file where the system will write the output in simple format (entailment
                decision and score) for each pair [optional]
-full=path      Path to file where the system will write the output in full format (entailment
                decision, score and edit operations) for each pair [optional]
-verbose=level     Verbose level of system messages: 0 (default) 1 2 [optional]
-overwrite      Overwrites the output files if they exist [optional]
```

As an example, the following lines represent two equivalent commands that run EDITS either using the ECF *conf1.xml*, or using parameters over the development set of RTE3 annotated at the morpho-syntax level. In both cases the execution of the command produces as output a distance threshold (i.e. the *distance model*) stored in the file *modelRTE3dev*. The *full* option indicates the file in which the system will save the edit operations performed by the system to transform **T** in **H**.

```
bin/edits -train rte/etaf/morpho-syntax/RTE3_dev.xml modelRTE3dev
    -algorithm=token
    -full=RTE3dev.out
    -scheme=share/cost-schemes/simple-scheme.xml

bin/edits -train -conf=share/configurations/conf1.xml rte/etaf/morpho-syntax/RTE3_dev.xml
modelRTE3dev
```

At the end of the training phase, a summary of the system's performance over the training set is printed on screen. This summary reports: (i) the distance model, including the distance threshold; (ii) the accuracy of the annotation of the whole training set; (iii) separate precision, recall and F-measure scores for the "YES" and "NO" training pairs.

```
Calculated Threshold
*******************************
0.7948717948717948
*******************************

Training Result:
*******************************
Accuracy: 0.61
*******************************
Entailment Class: YES
Number of Examples: 412
Precision: 0.6243781094527363
Recall: 0.6092233009708737
FMeasure: 0.6167076167076166
Classified as:
YES        251
NO        161
*******************************
Entailment Class: NO
Number of Examples: 388
Precision: 0.5954773869346733
Recall: 0.6108247422680413
FMeasure: 0.6030534351145037
Classified as:
YES        151
NO        237
*******************************
```

2.5 Test EDITS

Once a distance model is available, EDITS can be run over a test RTE file using the **bin/edit** command (with the **-test** option). The *input* parameter is the path of the test_set file, which has to be annotated in the ETAF format. In addition the **bin/edits** command allows for a number of options, reported below.

```
Usage: edits -test [-option(=value)?]* test_set

-conf=path      Configuration file path [optional]
-algorithm=name    Algorithm to use. Possible values: string (String Edit Distance -default) token
                (Token Edit Distance), tree (Tree Edit Distance) [optional]
-task=value     Calculate the threshold per task of the entailment pair: false (default) true
                [optional]
-length=value   Calculate the threshold per length of the entailment pair: false (default) true
                [optional]
-scheme=path    Path to a cost scheme file [optional]
-rules=path        Path to an entailment rules file [optional]
-simple=path    Path to file where the system will write the output in simple format (entailment
                decision and score) for each pair [optional]
-full=path      Path to file where the system will write the output in full format (entailment
                decision, score and edit operations) for each pair [optional]
-verbose=level  Verbose level of system messages: 0 (default) 1 2 [optional]
-overwrite      Overwrites the output files if they exist [optional]
-model=path        Model file path
```

As an example, the following two lines represent equivalent commands that run EDITS either using the ECF *conf1.xml*, or using command line options. In both cases the input is a corpus of RTE3 pairs annotated in the ETAF format at the morpho-syntax level, and the distance model is the *modelRTE3dev* created before.

```
bin/edits -test rte/etaf/morpho-syntax/RTE3_test.xml
    -algorithm=token
```

```
    -scheme=share/cost-schemes/simple-scheme.xml
    -model=modelRTE3dev

bin/edits -test -conf=share/configurations/conf1.xml -model=modelRTE3dev rte/etaf/morpho-syntax/
RTE3_test.xml
```

The XML Document Type Definition (DTD) of the output (as specified by the options **-full** and **-simple**) of EDITS is reported in Figure 2.

```
<!ELEMENT output (pair*,statistics?)>
<!ELEMENT pair (operations)>
<!ELEMENT operations (operation)>
<!ELEMENT operation EMPTY>
<!ELEMENT statistics (threshold+,elementClass+)>
<!ELEMENT threshold EMPTY>
<!ELEMENT elementClass (classifiedAs+)>
<!ELEMENT classifiedAs EMPTY>

<!ATTLIST pair
    id CDATA #REQUIRED
    entailment CDATA #REQUIRED
    original CDATA #REQUIRED
    score CDATA #REQUIRED
    confidence CDATA #REQUIRED
    distance CDATA #REQUIRED
    normalization CDATA #REQUIRED
    task CDATA #REQUIRED
>

<!ATTLIST operations cost CDATA #REQUIRED>
<!ATTLIST operation
    cost CDATA #REQUIRED
    type CDATA #REQUIRED
    source CDATA #IMPLIED
    target CDATA #IMPLIED
    scheme CDATA #REQUIRED
>

<!ATTLIST statistics accuracy CDATA #REQUIRED>

<!ATTLIST elementClass
    name CDATA #REQUIRED
    number CDATA #REQUIRED
    precision CDATA #REQUIRED
    recall CDATA #REQUIRED
    fmeasure CDATA #REQUIRED
>

<!ATTLIST classifiedAs
    name CDATA #REQUIRED
    number CDATA #REQUIRED
>
```

Fig. 2. DTD of EDITS output file

An example of RTE pair (from the RTE2 test set) annotated by the system is the following:

```
<pair task="IE" score="0.6470588235294118" original="YES" id="87" entailment="YES"
confidence="0.16421568627450975"/>
```

Beside the annotation of the entailment relation (i.e. *YES/NO*), for each pair the system provides the entailment score calculated by the algorithm and the confidence score of the entailment relation assignment (i.e. the distance from the threshold *S*).

At the end of the test phase a summary of the system's performance over the test set is printed on screen. This summary reports: (i) the accuracy of the annotation of the whole test set; (ii) separate precision, recall and F-measure scores for the "YES" and "NO" test pairs. For instance, the following is the evaluation made on the RTE2 test set using the configuration *conf1.xml*:

```
Test Result:
*******************************
Accuracy: 0.5525
*******************************
Entailment Class: YES
Number of Examples: 400
Precision: 0.54375
Recall: 0.6525
FMeasure: 0.5931818181818181
Classified as:
NO              139
YES             261
*******************************
Entailment Class: NO
Number of Examples: 400
Precision: 0.565625
Recall: 0.4525
FMeasure: 0.5027777777777779
Classified as:
NO              181
YES             219
*******************************
```

3 EDITS Text Annotation Format

The entailment corpus is represented in EDITS according to **ETAF** (EDITS Text Annotation Format), documented in *share/xsd/etaf.xsd*. The format is used both for representing the input **T**-**H** pairs and for representing entailment and contradiction rules. ETAF allows to represent texts at three different levels of annotation: (i) as simple strings; (ii) as sequences of tokens with their associated morpho-syntactic properties; (iii) as syntactic trees with structural relations among nodes. The three levels are considered in increasing complexity and levels of higher complexity assume the levels of lower complexity. For instance, texts annotated at syntactic level assume that the texts are also tokenized.

At each level a number of properties and their values can be introduced by the user through the use of the **attribute** element. For instance the part of speech of a certain token can be expressed by an attribute *pos* associated to a certain word element. Values of properties are retrieved by means of the **attribute** function.

The annotation is considered as a preliminary step before the actual use of the system, and both entailment pairs and entailment/contradiction rules must be defined according to this format.

The XML Document Type Definition (DTD) of the ETAF is reported in Figure 3.

3.1 ETAF String level

The first level of annotation represents texts as strings, i.e. as a sequence of characters, with their properties. As an example, the text "Edison invented the Kinetoscope." is simply represented as the sequence of its characters. This level is completely independent from any linguistic annotation.

```
<hAnnotation>
            <string>Edison invented the Kinetoscope.</string>
</hAnnotation>
```

```
<!ELEMENT entailment-corpus (pair+)>
<!ELEMENT pair (t, h, tAnnotation*, hAnnotation*)>
<!ELEMENT t (#PCDATA)>
<!ELEMENT h (#PCDATA)>
<!ELEMENT tAnnotation (string?,word*,tree?,semantics?)>
<!ELEMENT hAnnotation (string?,word*,tree?,semantics?)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT word (attribute+)>
<!ELEMENT attribute (#PCDATA)>
<!ELEMENT tree (node+,edge*)>
<!ELEMENT node (word|label)>
<!ELEMENT label (#PCDATA)>
<!ELEMENT edge EMPTY>
<!ELEMENT semantics (entity*,relation*)>
<!ELEMENT entity EMPTY>
<!ELEMENT relation EMPTY>

<!ATTLIST pair
    id CDATA #REQUIRED
    entailment (YES|NO|UNKNOWN) #REQUIRED
    task CDATA #IMPLIED
    length CDATA #IMPLIED
>
<!ATTLIST tAnnotation id CDATA #IMPLIED>
<!ATTLIST hAnnotation id CDATA #IMPLIED>
<!ATTLIST word id CDATA #IMPLIED>
<!ATTLIST attribute name CDATA #REQUIRED>
<!ATTLIST tree id CDATA #IMPLIED>
<!ATTLIST node id CDATA #IMPLIED>
<!ATTLIST edge
    name CDATA #IMPLIED
    from CDATA #REQUIRED
    to CDATA #REQUIRED
>
<!ATTLIST relation
    name CDATA #REQUIRED
    source CDATA #IMPLIED
>
<!ATTLIST entity
    name CDATA #REQUIRED
    start CDATA #IMPLIED
    end CDATA #IMPLIED
    source CDATA #IMPLIED
>
```

Fig. 3. DTD of the ETAF annotation format

3.2 ETAF Morpho-Syntax Level

The second level of annotation represents texts as sequences of tokens with morpho-syntactic features. Common properties are token, lemma, morpho and part of speech, but other linguistic annotations may be used at this level, including named entities, weights of tokens, like IDF, and many others.

For example, the following is the morpho-syntax representation of the sentence "Edison invented the Kinetoscope.", with user-defined attributes for two different pos tagging sets, token, lemma, full morphological analysis (full_morpho) and sentence boundaries.

```
<hAnnotation>
    <string>Edison invented the Kinetoscope.</string>
    <word id="2232">
        <attribute name="wnpos">n</attribute>
        <attribute name="token">Edison</attribute>
        <attribute name="lemma">edison</attribute>
        <attribute name="pos">NP0</attribute>
    </word>
    <word id="2233">
        <attribute name="full_morpho">invent+v+part+past invented+adj+zero
            invent+v+indic+past</attribute>
        <attribute name="wnpos">v</attribute>
        <attribute name="token">invented</attribute>
        <attribute name="lemma">invent</attribute>s
        <attribute name="pos">VVD</attribute>
    </word>
    <word id="2234">
        <attribute name="full_morpho">the+adv the+art</attribute>
        <attribute name="wnpos"></attribute>
        <attribute name="token">the</attribute>
        <attribute name="lemma">the</attribute>
        <attribute name="pos">AT0</attribute>
        </word>
    <word id="2235">
        <attribute name="wnpos">n</attribute>
        <attribute name="token">Kinetoscope</attribute>
        <attribute name="lemma">kinetoscope</attribute>
        <attribute name="pos">NN1</attribute>
    </word>
    <word id="2236">
        <attribute name="full_morpho">.+punc</attribute>
        <attribute name="sentence">&lt;eos&gt;</attribute>
        <attribute name="wnpos"></attribute>
        <attribute name="token">.</attribute>
        <attribute name="lemma">.</attribute>
        <attribute name="pos">PUN</attribute>
    </word>
</hAnnotation>
```

3.3 ETAF Syntactic Level

The third level of annotation represents texts as syntactic trees with their structural features. Both nodes (terminal and non terminal) and edges with syntactic relations are represented. Nodes are typically described with their morpho-syntactic properties. The example below shows the output of the Stanford Parser for the sentence "Edison invented the Kinetoscope." converted into ETAF.

```
<hAnnotation>
    <string>Edison invented the Kinetoscope.</string>
    <tree root="2">
        <node id="1">
            <word id="1">
                <attribute name="token">Edison</attribute>
                <attribute name="lemma">Edison</attribute>
                <attribute name="pos">NNP</attribute>
```

```
            </word>
        </node>
        <node id="2">
            <word id="2">
                <attribute name="token">invented</attribute>
                <attribute name="lemma">invent</attribute>
                <attribute name="pos">VBD</attribute>
            </word>
        </node>
        <node id="3">
            <word id="3">
                <attribute name="token">the</attribute>
                <attribute name="lemma">the</attribute>
                <attribute name="pos">DT</attribute>
            </word>
        </node>
        <node id="4">
            <word id="4">
                <attribute name="token">Kinetoscope</attribute>
                <attribute name="lemma">Kinetoscope</attribute>
                <attribute name="pos">NN</attribute>
            </word>
        </node>
        <edge to="1" name="nsubj" from="2"/>
        <edge to="3" name="det" from="4"/>
        <edge to="4" name="dobj" from="2"/>
    </tree>
</hAnnotation>
```

4 Using Edit Distance Algorithms

The EDITS package estimates the entailment relation among two text portions assuming that the entailment probability is inversely proportional with respect to the cost of transforming **T** into **H**. Such transformations are considered as edit operations (i.e. deletion, insertion and substitution) carried out over **T** and **H**. A main feature of EDITS is that it can use different edit distance algorithms, according to the linguistic annotation used to represent **T** and **H**. More specifically EDITS provides distance algorithms at three levels:

- *string edit distance* algorithms are used when **T** and **H** are represented as sequences of characters (i.e. strings) maintaining their original order;
- *token edit distance* algorithms are used when **T** and **H** are represented as sequences of tokens maintaining their original order;
- *tree edit distance* algorithms are used when **T** and **H** are represented according to their syntactic structure (e.g. dependency trees).

Different algorithms make use of different cost functions and different set of entailment rules. In the following sections the three classes of algorithms are detailed, as well as their requirements.

4.1 String Edit Distance

At this level, the three edit operations are defined over sequences of characters. Cost functions can only consider string properties of **T** and **H** (e.g. string length) and rules may refer to properties of characters (e.g. capitalization). This level of annotation is referred in ETAF as "string" level and is detailed in Section 3.1.

As a string edit distance algorithm, EDITS provides the Levenshtein distance algorithm [5], implemented in the class *StringEditDistance*.

4.2 Token Edit Distance

At this level, the three edit operations are defined over sequences of tokens of **T** and **H**. Cost functions may consider properties of tokenized sentences (e.g. sentence length) and rules may refer to morpho-syntactic properties of tokens, such as part of speech, lemma, morphological features and named entities. This level of linguistic annotation is referred in ETAF as "morpho-syntax" and is detailed in Section 3.2.

As a distance algorithm at the morpho-syntax level, EDITS provides a token-based version of the Levenshtein distance algorithm [5], implemented in the class *LinearEditDistance*.

4.3 Tree Edit Distance

At this level, the three edit operations are defined over single nodes of a syntactic representation of **T** and **H**. Cost functions may consider properties of syntactic trees (e.g. tree structure) and rules may refer to syntactic relations among nodes, such as dependency relations. This level of linguistic annotation is referred in EDITS as "syntax" and is detailed in Section 3.3.

As tree edit distance algorithm, the current version of EDITS implements the Zhang-Shasha algorithm [8] in the class *TreeEditDistance*. Since the Zhang-Shasha algorithm does not consider labels on edges, while dependency trees provide them, each dependency relation R from a node A to a node B has been re-written as a complex label $B-R$ concatenating the name of the destination node and the name of the relation. All nodes except the root of the tree are re-labeled in such way. Edit operations on single nodes are defined in the following way:

- **Insertion**: insert a node A from the dependency tree of **H** into the dependency tree of **T**. When a node is inserted it is attached with the dependency relation of the source label.
- **Deletion**: delete a node A from the dependency tree of **T**. When A is deleted all its children are attached to the parent of A. It is not required to explicitly delete the children of A, as they are going to be either deleted or substituted on a following step.
- **Substitution**: change the label of a node A in the source tree into a label of a node B of the target tree. In case of substitution the relation attached to the substituted node is changed with the relation of the new node.

4.4 How to configure an edit distance algorithm

EDITS allows to select a certain edit distance algorithm through a module declaration in the ECF. As an example, the declaration below (extracted from the pre-defined configuration file *conf2.xml* reported in Section 2.2) allows to use a tree edit distance algorithm.

```
<conf>

   <module type="entailment-engine">
          <!-- Active modules in the configuration -->
          <mlink idref="tree-edit-distance"/>
   </module>

   <module id="tree-edit-distance"
          type="distance-algorithm"
          className="eu.fbk.hlt.edits.distance.algorithms.TreeEditDistance"/>

</conf>
```

5 Defining Cost Schemes for Edit Operations

According to the distance-based approach, **T** entails **H** if there exists a sequence of transformations applied to **T** such that we can obtain **H** with an overall cost below a certain threshold. The underlying assumption is that pairs between which an entailment relation holds have a low cost of transformation.

EDITS allows for the definition of the cost for each edit operation carried out by the distance algorithm in order to find the best (i.e. less costly) sequence of edit operations that transforms **T** into **H**. The basic data structure in EDITS for the definition of costs is the *cost scheme*. One or more cost schemes can be associated to each edit operation, and they are collected in a *cost scheme file* that can be created by the user.

A cost scheme is invoked by the edit distance algorithm with three parameters: (i) an edit operation, (ii) an element of **T**, called the *source* and referred through the variable A, and (iii) an element of **H**, called the *target* and referred through the variable B. Each cost scheme for a certain edit operation consists of three parts:

1. **Name** - Every cost scheme must have a user defined unique name.
2. **Condition** - A set (possibly empty) of constraints over the source and the target elements, which need to be satisfied in order to activate the cost scheme. Each constraint is expressed in a lisp-like syntax, and all constraints must be satisfied (i.e. they have to return *true*) in order the cost scheme to be applied.
3. **Cost** - A fixed value, or a function that returns a numerical value, expressing the cost of the edit operation applied to the source and to the target. A cost function can consider as parameters the source element, the target element, the text **T**, and the hypothesis **H.**

EDITS adopts a combination of XML annotations and functional expressions to define the cost schemes. The XML Document Type Definition (DTD) of the cost scheme file is reported in Figure 4. As a simple example, EDITS provides a pre-defined cost scheme file (*simple-scheme.xml*, introduced in Section 2.2), where costs for the three edit operations are defined in the following way:

```
<schemes>
<scheme>
    <!-- A simple scheme for cost function calculation -->
    <!-- Valid constant types are : number boolean string and file -->
    <constant name="INSERTION" type="number" value="10"/>
    <constant name="DELETION" type="number" value="10"/>
    <constant name="SUBSTITUTION" type="number" value="20"/>
    <insertion name="insertion">
        <cost>INSERTION</cost>
    </insertion>
    <deletion name="deletion">
        <cost>DELETION</cost>
    </deletion>
    <substitution name="equal">
        <condition>(equals (attribute "token" A) (attribute "token" B))</condition>
        <cost>0</cost>
    </substitution>
    <substitution name="not-equal">
        <condition>(not (equals (attribute "token" A) (attribute "token" B)))</condition>
        <cost>SUBSTITUTION</cost>
    </substitution>
</scheme>
</schemes>
</schemes>
```

This cost scheme applies to elements of **T** and **H**, referred respectively as A and B, which are annotated as **word** (see ETAF in Section 3). The function (**attribute "token" A**) returns the token of the source element. Within the example, there are four cost schemes (1 for insertion, 1 for deletion, and 2 for substitution) expressing the following costs:

- insertion(B)= 10 - inserting an element B, no matter what B is, always costs 10.
- deletion(A)= 10 - deleting an element A, no matter what A is, always costs 10.
- substitution(A,B)= 0 *if* A=B - substituting A with B costs 0 if the token of A and the token of B are equal (i.e. they are the same string).
- substitution(A,B)= 20 *if* $A \neq B$ - substituting A with B costs 20 if the tokens of A and B are not equal.

Here below is a more complex example of a cost scheme for the substitution operation.

```
<substitution name="equal">
     <condition>(and (equals (attribute "lemma" A) (attribute "lemma" B))
                     (equals (attribute "pos" A) (attribute "pos" B)))
     </condition>
     <cost>0</cost>
</substitution>
```

In the example, a token from **T** is substituted by a token from **H** with a cost equal to 0 if their lemmas and part of speech are equal.

As shown in the previous examples, EDITS allows to define the cost of the edit operations by means of user-defined attributes. As an additional example, the following cost scheme exploits the pre-computed frequency of a token to calculate the cost of insertion, according to the intuition that the most frequent words should have a lower cost of insertion.

```
<insertion name="insertion_frequency">
    <condition>(not (null (attribute "freq" B)))</condition>
    <cost>(* (/ 1 (number (attribute "freq" B))) 20)</cost>
</insertion>
```

```
<!ELEMENT schemes (scheme+)>
<!ELEMENT scheme (constant*,insertion*,deletion*,substitution*)>
<!ELEMENT constant EMPTY>
<!ELEMENT insertion (condition*,cost)>
<!ELEMENT deletion (condition*,cost)>
<!ELEMENT substitution (condition*,cost)>

<!ELEMENT condition (#PCDATA)>
<!ELEMENT cost (#PCDATA)>

<!ATTLIST scheme name CDATA #REQUIRED>

<!ATTLIST insertion name CDATA #REQUIRED>
<!ATTLIST deletion name CDATA #REQUIRED>
<!ATTLIST substitution name CDATA #REQUIRED>

<!ATTLIST constant
    name CDATA #REQUIRED
    type (string|number|boolean) #REQUIRED
    value CDATA #REQUIRED
>
```

Fig. 4. DTD of cost scheme file

5.1 Data Types and Functions

Conditions and costs are defined using a set of functions, expressed in a lisp-like syntax. Such functions can consider as parameters the source A, the target B, the text **T**, and the hypothesis **H**. This means that all the information about **T** and **H** derived from their linguistic processing (e.g. part of speech, syntactic structure, etc.) is available for defining conditions and cost functions. As an example, typical constraints involve checking the token and the part of speech of A and B, while typical cost functions are computed considering the lexical similarity between A and B, possibly normalizing such value over the length of **T** and **H**.

5.1.1 Data Types

Basic elements for defining constraints and costs in a cost scheme are derived from the three representation levels defined in ETAF (see Section 3). EDITS provides functions for the most relevant elements defined in the ETAF linguistic representation. The arguments of such functions are the variables (i.e. A, B, **T**, **H**) which are instantiated within a specific cost scheme.

Functions use the following primitive objects data types, hierarchically arranged in Figure 5:

1. **Word** - represents a token in **T** and **H** and it is instantiated by the variables A and B in a cost scheme.
2. **Node** - represents a tree element in **T** and **H** and it is instantiated by the variables A and B in a cost scheme.
3. **Tree** - represents a syntactic tree; it is obtained using the function **tree**.

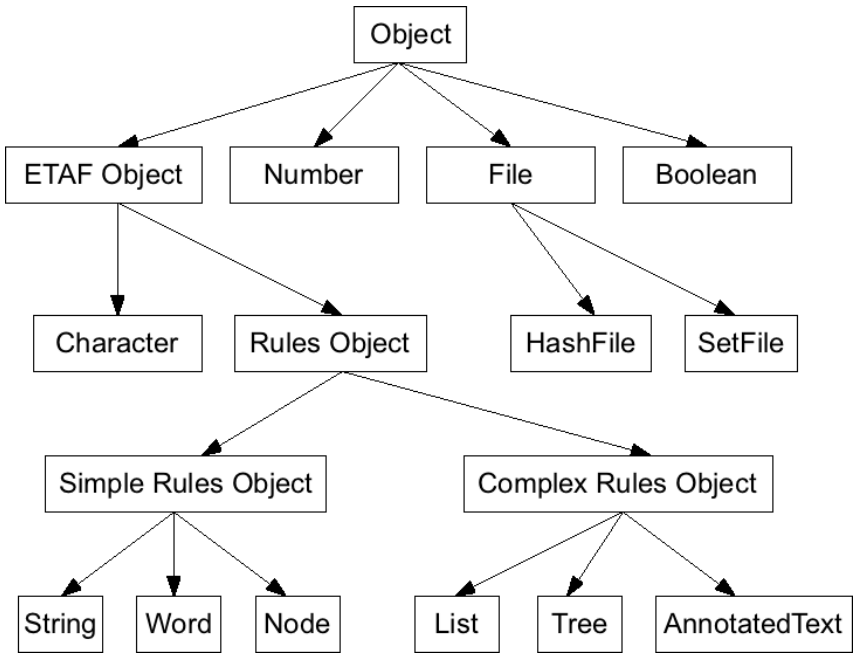

Fig. 5. EDTIS datatype hierarchy

4. **Number** - a real number, for example: 0 , 3.14 etc.
5. **Boolean** - True | False.
6. **String** - a sequence of characters, for example: "Dolomiti" or "Milen".
7. **Char** - a symbol, for example: 'a' or '?'.
8. **List** - a sequence of elements (words, nodes, numbers, booleans, etc.).
9. **SetFile** - a group of elements of a given type (strings, numbers, or booleans) loaded from a file .
10. **HashFile** - an object that contains maps from keys to values, loaded from a file. Keys are strings, while values are either strings, numbers, or booleans.
11. **AnnotatedText** - represents the annotated text of **T** or **H** accessible using the variables T and H.

SetFiles and HashFiles are objects that have to be read from an external file (i.e. can not be created inside a cost scheme). The format of a HashFile is shown in Figure 6. The *type* is the data type of the *values* in the HashFile, and the *key* is separated from a *value* with a tab. An example of a HashFile file containing the IDF of words is shown in Figure 7.

```
type
key-1    value-1
...
key-n    value-n
```

Fig. 6. HashFile format

```
number
speak  12.23
ride   3.23
...
read   10.32
```

Fig. 7. HashFile example

The format of a SetFile is shown in Figure 8. The *type* is the data type of the elements in the SetFile. An example of a SetFile file containing stop words is shown in Figure 9.

```
type
value-1
value-2
value-3
```

Fig. 8. SetFile format

```
string
a
a's
able
about
above
according
accordingly
```

Fig. 9. SetFile example

The system also treats the string *null* as expressing a non-existing value.

5.1.2 Functions

Functions over AnnotatedText

- (**string** *AnnotatedText*) - returns the text of the AnnotatedText (i.e. the text of **T** or **H**).
- (**tree** *AnnotatedText*) - returns the syntactic tree of the AnnotatedText.
- (**words** *AnnotatedText*) - returns the list of words in the AnnotatedText.

Functions for accessing Entailment Rules

- (**entail** $SimpleRulesObject_1$ $SimpleRulesObject_2$) - checks for the existence of an entailment rule (see Section 6) where the left hand side of the rule matches $SimpleRulesObject_1$ and the right hand side of the rule matches $SimpleRulesObject_2$. The two arguments must be of the same data type. The allowed types are: String, Word and Node. If the rule exists, then the probability associated to the rule is returned, otherwise the output of the function is null.
- (**contradict** $SimpleRulesObject_1$ $SimpleRulesObject_2$) - checks for the existence of a contradiction rule (see Section 6) where the left hand side of the rule matches $SimpleRulesObject_1$ and the right hand side of the rule matches $SimpleRulesObject_2$. The two arguments must be of the same data type. The allowed types are: String, Word and Node. If the rule exists, then the probability associated to the rule is returned, otherwise the output of the function is null.

Functions over Trees

- (**nodes** *Tree*) - returns the list of nodes of a tree.
- (**parent** *Node Tree*) - returns the parent of a node in the syntactic tree of **T** or **H**.
- (**children** *Node Tree*) - returns the children of a node in the syntactic tree of **T** or **H**.

Functions over Nodes

- (**word** *Node*) - returns the word of the node.
- (**label** *Node*) - returns the label (i.e. the syntactic category) of the node.
- (**edge** *Node*) - returns the edge (i.e. the syntactic relation) entering in the node.
- (**is-label-node** *Node*) - returns true if the node contains a label.
- (**is-word-node** *Node*) - returns true if the node contains a word.

Functions over Words

– (**attribute** *String Word*) - returns the value of the attribute *String* of the word. If the attribute is missing the function returns null.

Functions with string arguments

– (**equals** $String_1$ $String_2$) - Returns True if $String_1$ is equal to $String_2$.
– (**equals-ignore-case** $String_1$ $String_2$) - compares two strings ignoring their case.
– (**capitalized** *String*) - returns true if the string is capitalized.
– (**starts-with** $String_1$ $String_2$) - returns True if $String_1$ starts with $String_2$. For instance: (**starts-with** *"reading" "read"*) is True.
– (**ends-with** $String_1$ $String_2$) - returns *True* if $String_1$ ends with $String_2$.
– (**contains** $String_1$ $String_2$) - returns *True* if $String_1$ contains $String_2$. For instance, (**contains** *"reenacting" "act"*) returns True.
– (**number** *String*) - reads a number from *String*. For instance, (**number** *"3.14"*) returns 3.14.
– (**boolean** *String*) - reads a boolean from *String*. The possible arguments are *"true"* and *"false"*.
– (**to-lower-case** *String*) - converts *String* to lower case.
– (**length** *String*) - returns the number of characters in *String*.
– (**char** *String Number*) - returns the character in *String* at position corresponding to *Number*.
– (**substring** *String* $Number_1$ $Number_2$) - returns the sub-string of *String* from the position corresponding to $Number_1$ till the position $Number_2$.
– (**distance** $String_1$ $String_2$ *:normalize*) returns the Levenshtein distance between $String_1$ and $String_2$. If the ":normalize" parameter is present the function returns a normalized distance (with respect to the length of the two arguments) between 0 and 1.

Functions with numeric arguments

– $(= Number_1\ Number_2)$ - returns *True* if $Number_1$ is equal to $Number_2$.
– $(< Number_1\ Number_2)$ - returns *True* if $Number_1$ is less than $Number_2$.
– $(> Number_1\ Number_2)$ - returns *True* if $Number_1$ is more than $Number_2$.
– $(+\ Number_1 \ldots Number_n)$ - makes a sum of numbers (example (+ 1 2) is equal to 3).
– $(-\ Number_1 \ldots Number_n)$ - subtracts numbers from $Number_1$ (example (- 5 2 1) is equal to 2).
– $(*\ Number_1 \ldots Number_n)$ - multiplies numbers (example (* 2 2) is equal to 4).
– $(/\ Number_1 \ldots Number_n)$ - divides $Number_1$ by the rest (example (/ 24 3 2) is equal to 4).

Functions with boolean arguments

– (**and** $Boolean_1 \ldots Boolean_n$) - returns *True* if all the arguments are *True*.
– (**or** $Boolean_1 \ldots Boolean_n$) - returns *True* if at least one of the arguments is *True*.
– (**not** *Boolean*) - returns *True* if the argument is *False*, and *False* if it is *True*.

Conditional Functions

– (**if** *Boolean* $Object_1$ $Object_2$) - if the *Boolean* is equal to *True* then the function returns $Object_1$ otherwise $Object_2$. If $Object_2$ is not defined the function returns *null*.

Functions with list arguments

– (**member** *List Object*) - returns *True* if *List* contains *Object*. For example: the list (1 2 3) contains the number 1; the list ("sum" "plus" "minus") contains the string "plus".
– (**size** *List*) - returns the number of elements in *List*.
– (**nth** *Number List*) - returns the *n-th* (Number) element of *List*. The first element of a list is returned by (nth 1 list), the last with (nth (- (size list) 1) list).
– (**subseq** *List* $Number_1$ $Number_2$)- returns the sub-list of the list from the position corresponding to $Number_1$ till the position $Number_2$.

Functions handling HashFiles and SetFiles

– (**make-hash-file** *String*) - Reads a *HashFile* from the path specified in *String*.
– (**make-set-file** *String*) - Reads a *SetFile* from the path specified in *String*.
– (**hash-value** *HashFile String*) - returns the value of the map inside *HashFile* with key *String*.
– (**set-contains** *HashFile Object*) - returns *True* if *SetFile* contains *Object*.

Null handling functions

– (**null** *Object*) - returns *True* if the argument is *null*. For example, to express that a word *A* from **T** has not an attribute "freq", the expression (**null** (*attribute "freq" A*)) can be used.

5.2 How to configure cost schemes

The cost scheme file is defined in the ECF. As an example, the declaration below (extracted from the pre-defined configuration file *conf1.xml* in Section 2.2) allows to use the *simple-scheme.xml*.

```
<conf>
   <module type="entailment-engine">
           <mlink idref="xml-cost-scheme"/>
   </module>

   <module id="xml-cost-scheme"
           type="cost-scheme"
           className="eu.fbk.hlt.edits.distance.cost.scheme.XmlCostScheme">
           <attribute name="scheme-file">${EDITS_PATH}/share/cost-schemes/simple-scheme.xml</attribute>
   </module>
</conf>
```

6 Defining rules in EDITS

EDITS allows the use of sets of *rules*, both *entailment rules* and *contradiction rules*, in order to provide specific knowledge (e.g. lexical, syntactic, semantic) about transformations between **T** and **H**. Rules can be manually produced, or they can be extracted from any available resource (e.g. WordNet, Wikipedia, DIRT) and stored in XML files which are called *Rule Repositories*.

Each rule in EDITS consists of four parts:

1. **rule name** - a unique identifier of the rule within a certain rule repository. This is used for logging purposes only, in order to help the user to understand which rules have been applied by the system for a certain pair. If not provided by the user, the rule name is automatically generated by the system.
2. **t** - a text **T**, i.e. the left hand side of the rule.
3. **h** - a hypothesis **H**, i.e. the right hand side of the rule.
4. **rule probability** - a probability that the rule maintains either the entailment or the contradiction between **T** and **H**. Both in entailment and contradiction rules, a probability equal to 0 means that the relation between **T** and **H** is unknown, while a probability equal to 1 means that the entailment/contradiction between **T** and **H** is fully preserved.

6.1 Rule format

Both **T** and **H** can be defined using the Edits Text Annotation Format (ETAF) described in Section 3. ETAF allows text portions to be represented at three different levels of annotation: just as strings (i.e. the STRING object), as sequences of tokens with their morpho-syntactic features (i.e. the WORD object), and as syntactic trees (e.g. the NODE object). Rules in EDITS can be defined using the three datatypes, provided that they are used consistently in **T** and **H**, i.e. either STRING, or WORD or NODE. The XML Document Type Definition (DTD) of the rules file is reported in Figure 10.

In the current release of EDITS only rules that contain just one element both in **t** and **h** (i.e. lexical rules) are allowed.

6.1.1 Entailment rules. Entailment rules preserve, with some degree of confidence, the entailment relation between **T** and **H**. The following are examples of entailment rules at the different levels allowed by ETAF.

String level:

```
<rule name="1">
    <t><string>invented</string></t>
    <h><string>pioneered</string></h>
    <probability>1.0</probability>
</rule>
```

```
<!ELEMENT rules (rule+)>
<!ELEMENT rule (t, h,probablity)>
<!ELEMENT t (string?,word*,tree?,semantics?)>
<!ELEMENT h (string?,word*,tree?,semantics?)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT word (attribute+)>
<!ELEMENT attribute (#PCDATA)>
<!ELEMENT tree (node+,edge*)>
<!ELEMENT node (word|label)>
<!ELEMENT label (#PCDATA)>
<!ELEMENT edge EMPTY>
<!ELEMENT semantics (entity*,relation*)>
<!ELEMENT entity EMPTY>
<!ELEMENT relation EMPTY>
<!ATTLIST rule name CDATA #REQUIRED>
<!ATTLIST t id CDATA #IMPLIED>
<!ATTLIST h id CDATA #IMPLIED>
<!ATTLIST word id CDATA #IMPLIED>
<!ATTLIST attribute name CDATA #REQUIRED>
<!ATTLIST tree id CDATA #IMPLIED>
<!ATTLIST node id CDATA #IMPLIED>
<!ATTLIST edge
    name CDATA #IMPLIED
    from CDATA #REQUIRED
    to CDATA #REQUIRED
>
<!ATTLIST relation
    name CDATA #REQUIRED
    source CDATA #IMPLIED
>
<!ATTLIST entity
    name CDATA #REQUIRED
    start CDATA #IMPLIED
    end CDATA #IMPLIED
    source CDATA #IMPLIED
>
```

Fig. 10. DTD of the rule file

The entailment rule above states that the string *invented* entails the string *pioneered* with a probability equal to 1.0.

Morpho-Syntax level:

```
<rule name="2">
    <t><word><attribute name="lemma">invent<attribute></word></t>
    <h><word><attribute name="lemma">pioneer<attribute></word></h>
    <probability>1.0</probability>
</rule>
```

The entailment rule above states that the lemma *invent* entails the lemma *pioneer* with a probability equal to 1.0.

Syntactic level:

```
<rule name="3">
    <t><node><attribute name="edge-to-parent">dobj</attribute>
       <word><attribute name="token">home<attribute></word></node></t>
    <h><node><attribute name="edge-to-parent">dobj</attribute>
       <word><attribute name="token">habitation<attribute></word></node></h>
    <probability>1.0</probability>
</rule>
```

The entailment rule above states that the node *home* in the dependency relation of direct object with its syntactic head (e.g. "John bought a *house*", where *house* is the direct object of the verb *to buy*) entails the node *habitation* in the dependency relation of direct object with its syntactic head (e.g. "John bought an *habitation*", where *habitation* is the direct object of the verb *to buy*) with a probability equal to 1.0.

6.1.2 Contradiction rules. Contradiction rules represent, with some degree of confidence, the semantic incompatibility between **T** and **H**. The following are examples of contradiction rules at the different levels allowed by ETAF.

String level:

```
<rule name="a">
    <t><string>beautiful</string></t>
    <h><string>ugly</string></h>
    <probability>1.0</probability>
</rule>
```

The contradiction rule above states that the string *beautiful* contradicts the string *ugly* with a probability equal to 1.0.

Morpho-Syntax level:

```
<rule name="b">
    <t><word><attribute name="token">extend<attribute></word></t>
    <h><word><attribute name="token">shorten<attribute></word></h>
    <probability>1.0</probability>
</rule>
```

The contradiction rule above states that the lemma *extend* contradicts the lemma *shorten* with a probability equal to 1.0.

Syntactic level:

```
<rule name="c">
    <t><node><attribute name="edge-to-parent">amod</attribute>
       <word><attribute name="token">white<attribute></word></node></t>
    <h><node><attribute name="edge-to-parent">amod</attribute>
       <word><attribute name="token">black<attribute></word></node></h>
    <probability>1.0</probability>
</rule>
```

The contradiction rule above states that the node *white* in the dependency relation of adjectival modifier with its syntactic head (e.g. "Mary wears a *white* T-shirt", where *white* is the adjective modifying the noun *T-shirt*) contradicts the node *black* in the dependency relation of adjectival modifier with its syntactic head (e.g. "Mary wears a *black* T-shirt", where *black* is the adjective modifying the noun *T-shirt*) with a probability equal to 1.0.

6.2 Rules repository

In order to be used by EDITS, both entailment and contradiction rules have to be stored in a rule repository. EDITS allows to declare and use multiple XML rule files as sets of entailment or contradiction rules that can be refereed using user defined identifiers. As an example, the declaration below defines a rule repository called *wordnet-rules*, which contains two rule files, called, respectively, *entailment-rules-wordnet* and *contradiction-rules-wordnet*. An example of entailment rule file can be found in `edits/share/rules`.

```
<conf>

   <module type="entailment-engine">
          <!-- Active modules in the configuration -->
          <mlink idref="wordnet-rules"/>
   </module>

   <module id="wordnet-rules"
          type="rules-repository"
          className="eu.fbk.hlt.edits.rules.DefaultRulesRepository">
          <option name="entailment-rules" id="entailment-rules-wordnet">
          ${EDITS_PATH}/share/rules/entailment-rules-wordnet.xml
          </option>
          <option name="contradiction-rules" id="contradiction-rules-wordnet">
          ${EDITS_PATH}/share/rules/contradiction-rules-wordnet.xml
          </option>
   </module>
</conf>
```

6.3 Rule activation

The basic way for activating a rule is through one of the two functions, **entail** and **contradict**, which can be used within a cost scheme (see Section 5). The two functions check for the existence of an entailment or a contradiction rule between the values assumed by A and B in a cost schema. If a rule exists in the specified repository which matches with both A and B, then the probability associated to the rule is returned, otherwise null. The two functions accept four parameters:

1. The first two parameters X and Y are portions of **T** and **H** managed by the distance algorithm and the cost scheme.
2. The name of a set of rules in the rules repository where the search has to be carried on. This parameter is optional.
3. The search modality. Two search modalities are allowed: **First**, which selects the first rule that matches the X and Y parameters; **Max**, which selects the rule that matches X and Y and with the highest probability. The search modality is cab also be declared, as a default strategy, at the level of configuration file.

   As an example, the following call at the **entail** function:

   ```
   (entail X Y "entailment-rules-wordnet" :max)
   ```

   searches for the rule with the highest probability among those that are activated by the A and B parameters and that are contained in a rules set in the rules repository called *entailment-rules-wordnet*.

A rule is activated when the X parameter of the **entail**/**contradiction** function matches with the **T** part of the rule, i.e. the left hand side, and the Y parameter of the function matches with the **H** part of the rule, i.e. the right hand side. All the elements of the X/Y argument have to match against all

elements of the rule. In case the rule contains variables, their assignments to corresponding elements of the X/Y argument need to be satisfied.

The **entail** and **contradict** functions are called in cost schemes, typically the cost scheme defined for the substitution edit operation. As an example, the following is a cost scheme that calculates the cost of substituting A with B based on the inverse of the probability of the entailment rule between A and B in the repository called *entailment-rules-wordnet*.

```
<substitution name="entail">
  <cost>(- 1 (entail A B :entailment-rules-wordnet :max))</cost>
</substitution>
```

7 EDITS configuration file

The purpose of a configuration file is to define the three modules (i.e. distance algorithm, cost scheme and rule repositories), and their corresponding parameters, that will be actually used while running EDITS on a certain dataset. Only modules defined in a EDITS Configuration File (ECF) can be used for training and testing with the command **bin/edits** (see Sections 2.4 and 2.5).

A module may require that another module is defined in order to work. Such dependencies are expressed in a configuration file through nested modules. The whole EDITS configuration is considered itself a module, the more global one, called the **entailment engine**, which requires three nested modules, respectively for the distance-algorithm, the cost-scheme and the rules-repository.

The XML Document Type Definition (DTD) of the configuration file is reported in Figure 11, while the following is a simple configuration file, where the whole EDITS is defined as a module of type entailment engine, which, in turn, needs three modules, referred with through their identifiers in the *mlink* elements.

```
<!ELEMENT conf (constant* module*)>
<!ELEMENT module (module*,attribute*,option*, mlink*)>
<!ELEMENT attribute (#PCDATA)>
<!ELEMENT option (#PCDATA)>
<!ELEMENT mlink (#PCDATA)>


<!ATTLIST module
    type CDATA #IMPLIED
    id CDATA #IMPLIED
    className CDATA #IMPLIED
>

<!ATTLIST attribute
    name CDATA #REQUIRED
    id CDATA #IMPLIED
>
<!ATTLIST option
    name CDATA #REQUIRED
    id CDATA #IMPLIED
>

<!ATTLIST mlink
    idref CDATA #REQUIRED
>
```

Fig. 11. DTD of the configuration file

```
<conf>

   <module type="entailment-engine">
```

```
        <mlink idref="tree-edit-distance"/>
        <mlink idref="xml-cost-scheme"/>
        <mlink idref="entailment-rules-wordnet"/>
    </module>

    <module id="tree-edit-distance"
        type="distance-algorithm"
        className="eu.fbk.hlt.edits.distance.algorithms.TreeEditDistance"/>

    <module id="xml-cost-scheme"
        type="cost-scheme"
        className="eu.fbk.hlt.edits.distance.cost.scheme.XmlCostScheme">
        <attribute name="scheme-file">${EDITS_PATH}/share/cost-schemes/simple-scheme.xml
        </attribute>
    </module>

    <module id="entailment-rules-wordnet"
        type="rules-repository"
        className="eu.fbk.hlt.edits.rules.DefaultRulesRepository">
        <option name="entailment-rules">${EDITS_PATH}/share/rules/entailment-rules-wordnet.xml
        </option>
    </module>

</conf>
```

7.1 Module Configuration

Modules are defined by the following pieces of information:

- *className* attribute: a path to the Java class of the module, referring to the code that will be executed when the module is activated;
- *id* attribute: a unique identifier for the module, assigned by the user;
- *type* attribute: indicates the category of the module being defined. Accepted values for the type attribute are entailment-engine, distance-algorithm, cost-scheme, rules-repository. Such types are also accepted by the **bin/edits** command;
- *attribute* element: used to set the parameters of the module;
- *option* element: used to set the options of the module;
- *mlink* element: links to nested modules, i.e. submodules that are required by the module being defined.

As an example, in addition to those presented in the previous Section, the following is the configuration of a module defining the use of TextPro, an interface to a text annotator offered by EDITS (see Section 2.3). The module is defined by the *id*, its *class path*, and a number of parameters that will be used to run TextPro.

```
<module id="textpro"
        className="eu.fbk.hlt.annotation.textpro.TextPro">
        <attribute name="script">/home/epack/scripts/textpro</attribute>
        <attribute name="tempDir">/home/epack/temp/</attribute>
        <attribute name="tempFile">question-pure</attribute>
        <attribute name="extension">txp</attribute>
        <attribute name="encoding">Latin1</attribute>
        <attribute name="language">Italian</attribute>
</module>
```

7.2 Usage of Constants

EDITS allows that the values of the attributes (which are frequently used in the configuration file) can be referred through the use of constants, declared at the beginning of the configuration file.

For example, the TextPro configuration module showed above, could be re-written with the use of a constant in the following way, so that the location of TextPro can be defined only once and can be easily changed if the package is later relocated.

```
<conf>

   <constant name="PATH">/home/epack/edits</constant>
   <module id="textpro"
           type="eu.fbk.hlt.annotation.textpro.TextPro">
           <attribute name="script">$VAR{PATH}/scripts/textpro</attribute>
           <attribute name="tempDir"> $VAR{PATH}/ temp/</attribute>
           <attribute name="tempFile">question-pure</attribute>
           <attribute name="extension">txp</attribute>
           <attribute name="encoding">Latin1</attribute>
           <attribute name="language">Italian</attribute>
  </module>

</conf>
```

7.3 In-line and out-line definitions

A configuration file can be defined in two modalities: (i) *in-line*, when modules are nested one into the other without making use of *mlinks* to other modules; (ii) *out-line*, when module identifiers are used to refer to such modules.

The *in-line* modality produces a file describing all necessary modules, while the *out-line* modality allows to define in a single file more variants for the same EDITS functionality, and to activate just one of them using a mlink. As an example of *out-line* configuration, the following file shows that in a single file, while more than one module for the same function (i.e. *distance-algorithm*) can be defined (i.e. *token-edit-distance*, *string-edit-distance* and *tree-edit-distance*), just one of them (i.e. *tree-edit-distance*) is used as nested by the entailment-engine and will be actually activated while running the system. In order to change the module to be activated, the *idref* of the corresponding *mlink* should be simply substituted.

```
<conf>

   <module type="entailment-engine">
           <mlink idref="tree-edit-distance"/>
           <mlink idref="xml-cost-scheme"/>
           <mlink idref="entailment-rules-wordnet"/>
   </module>

   <module id="token-edit-distance"
           type="distance-algorithm"
           className="eu.fbk.hlt.edits.distance.algorithms.TokenEditDistance"/>

   <module id="string-edit-distance"
           type="distance-algorithm"
           className="eu.fbk.hlt.edits.distance.algorithms.StringEditDistance"/>

   <module id="tree-edit-distance"
           type="distance-algorithm"
           className="eu.fbk.hlt.edits.distance.algorithms.TreeEditDistance"/>

   <module id="xml-cost-scheme"
           type="cost-scheme"
           className="eu.fbk.hlt.edits.distance.cost.scheme.XmlCostScheme">
           <attribute name="scheme-file">${EDITS_PATH}/share/cost-schemes/simple-scheme.xml
           </attribute>
   </module>

   <module id="entailment-rules-wordnet"
           type="rules-repository"
           className="eu.fbk.hlt.edits.rules.DefaultRulesRepository">
           <option name="entailment-rules">${EDITS_PATH}/share/rules/entailment-rules-
           wordnet.xml</option>
   </module>
```

```
</conf>
```

It is also possible to have a mixed configuration, mixing both *in-line* and *out-line* definitions.

7.4 Pre-defined configurations

EDITS provides few pre-defined configurations, aiming at facilitating the use of the system.

Configuration 1. This configuration (*conf1.xml*) makes use of the following modules:

- *distance algorithm*: token distance algorithm, i.e. Levenshtein distance [5], calculated over the tokens of **T** and **H**.
- *cost scheme*: a cost scheme where the costs of each edit operation are fixed.
- *rules*: no rules are used in this configuration
- *linguistic annotation*: **T** and **H** are annotated using TextPro.

The configuration has been experimented using the RTE1, RTE2 and RTE3 datasets for training and RTE-4 for testing.

Configuration 2. This configuration makes use of the following modules:

- *distance algorithm*: tree distance algorithm, i.e. Zhang-Shasha [8], calculated over the nodes of the dependency tree of **T** and **H**.
- *cost scheme*: a cost scheme where the costs of each edit operation are dynamically calculated considering the length of **T** and **H**.
- *rules*: a repository of entailment rules, automatically extracted from WordNet. Rules are built according to the following procedure: for all pairs in the RTE collection (i.e. all RTE training and test data available so far), and for all combination of tokens A in **T** and tokens B in **H**, we seek in WordNet (version 3.0) whether the following relations holds: $Synonym(A, B)$, $Hypernym(A, B)$, $Hypernym(B, A)$. If the relation holds, then an entailment rule is built between A and B with probability 1.
- *linguistic annotation*: **T** and **H** are annotated using the Stanford parser.

The configuration has been experimented using the RTE1, RTE2 and RTE3 datasets for training and RTE-4 for testing.

References

1. Ido Dagan, Oren Glickman (2004), *Probabilistic Textual Entailment: Generic Applied Modeling of Language Variability*, in *Proceedings of the PASCAL Workshop of Learning Methods for Text Understanding and Mining*, Grenoble, France.
2. Ido Dagan, Oren Glickman, Bernardo Magnini (2005), *The PASCAL Recognizing Textual Entailment Challenge*, in *Proceedings of the First PASCAL Challenges Workshop on Recognising Textual Entailment*, Southampton, U.K., 11-13 April.
3. Dan Klein and Christopher D. Manning (2003), *Fast Exact Inference with a Factored Model for Natural Language Parsing*, in *Advances in Neural Information Processing Systems 15 (NIPS 2002)*, Cambridge, MA: MIT Press, pp. 3-10.
4. Milen Kouylekov, Bernardo Magnini (2005), *Tree Edit Distance for Recognizing Textual Entailment*, in *Proceedings of the International Conference Recent Advances in Natural Language Processing (RANLP)*, Borovets, Bulgaria, 21-23 September.
5. Vladimir I. Levenshtein (1965), *Binary codes capable of correcting deletions, insertions, and reversals*, in *Doklady Akademii Nauk SSSR, 163(4)*, pages 845848.
6. Dekang Lin (1998), *Dependency-based Evaluation of MINIPAR*, in *Workshop on the Evaluation of Parsing Systems*, Granada, Spain, May.
7. Emanuele Pianta, Christian Girardi, Roberto Zanoli (2008), *The TextPro tool suite*, in *Proceedings of LREC, 6th edition of the Language Resources and Evaluation Conference*, Marrakech, Morocco, 28-30 May.
8. Kaizhong Zhang, and Dennis Shasha (1990). Fast Algorithm for the Unit Cost Editing Distance Between Trees. In *Journal of Algorithms*. vol.11, December 1990.