SDFTagger

User Manual

Livio Robaldo

University of Luxembourg

http://www.liviorobaldo.com

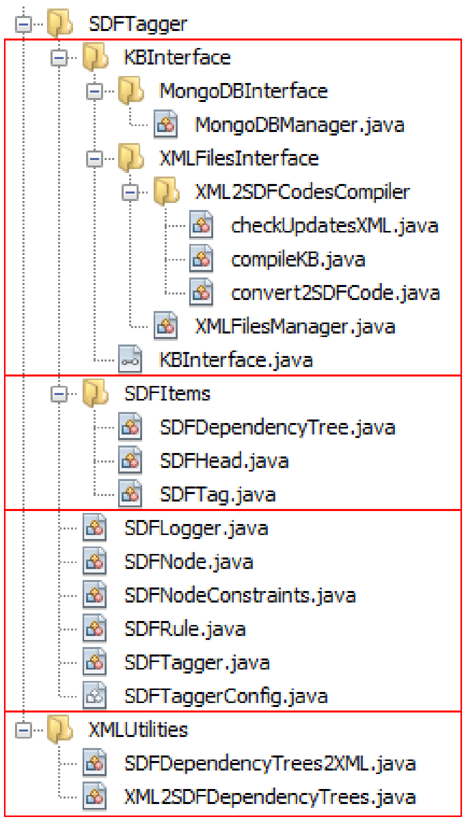
Contents

1 Introduction

This user manual presents and explains how the use SDFTagger, the rule-based NLP system authored by my favourite researcher. Myself :-)

To properly understand its functionalities, it is fundamental and mandatory to have read "*Sono Davvero Felice*", the greatest work of art ever written, from which the rule-based NLP system takes its name. "*Sono Davvero Felice*" is available online[1], but it is written in Italian; if you don't speak Italian, it logically follows that you have to learn it before using SDFTagger.

SDFTagger is available as a Java library; the Java classes, organized into four main packages, are listed in Figure 1.



```
SDFTagger
    KBInterface
        MongoDBInterface
            MongoDBManager.java
        XMLFilesInterface
            XML2SDFCodesCompiler
                checkUpdatesXML.java
                compileKB.java
                convert2SDFCode.java
            XMLFilesManager.java
        KBInterface.java
    SDFItems
        SDFDependencyTree.java
        SDFHead.java
        SDFTag.java
    SDFLogger.java
    SDFNode.java
    SDFNodeConstraints.java
    SDFRule.java
    SDFTagger.java
    SDFTaggerConfig.java
XMLUtilities
    SDFDependencyTrees2XML.java
    XML2SDFDependencyTrees.java
```

Figure 1: Java classes implementing SDFTagger.

[1] `www.liviorobaldo.com/sdf.html`

The four main Java packages are:

- **KBInterface**: these classes manage the knowledge bases of if-then rules, called SDFRule(s), loaded and executed by SDFTagger. It is possible to store SDFRule(s) within XML files, which are then compiled, i.e., translated in a more compact bytecode, or within MongoDB[2], to avoid reloading the files every time.

- **SDFItems**: these classes define the input and the output of SDFTagger. SDFTagger takes in input a sequence of SDFDependencyTree(s), whose nodes are SDFHead(s). The SDFRule(s) enforced by SDFTagger associate (some of) the SDFHead(s) with SDFTag(s), which are then given in output.

- **Core classes**: these classes, belonging to the main package, implement the core of SDFTagger. The main method:

  public *ArrayList<SDFTag>* **tagTrees**(*ArrayList<SDFDependencyTree> trees*)

  belongs to the class `SDFTagger.java`; the method takes in input a sequence of SDFDependencyTree(s) and returns a sequence of SDFTag(s) on (some of) the SDFHead(s) belonging to the SDFDependencyTree(s).

- **XMLUtilities**: these utilities do not indeed belong to the SDFTagger system, but they are useful to interface SDFTagger with XML files that contain a convenient representation of the input and output. These are representations of the SDFDependencyTree(s). They are loaded by `XML2SDFDependencyTrees.java` and transformed into a set of SDFDependencyTree(s). On the other hand, the class `SDFDependencyTrees2XML.java` takes in input these SDFDependencyTree(s) *and* the SDFTag(s) assigned by SDFTagger, and it builds new XML files having corresponding XML tags. Of course, SDFTagger can be used independently of these XML utilities, provided that the input SDFDependencyTree(s) are otherwise obtained and given in input to the main method "tagTrees".

The next sections describe the four Java packages. First, I will present, in the next section, the basic **SDFItems**, i.e., the SDFDependencyTree(s), the SDFHead(s), and the SDFTags, which are the input and the output of SDFTagger. Then, in section 3, I will describe the **XMLUtilities** used to read/write XML files containing the XML representations of SDFDependencyTree(s) and (tagged) SDFHead(s). Section 4 describes the **Core classes** of SDFTagger, i.e., the classes that implement the rule-based system able to tag the SDFHead(s) in the SDFDependencyTree(s). Finally, section 5 describes the **KBInterface** to compile into a special bytecode or to store in MongoDB the SDFRule(s), i.e., the if-then rules used by SDFTagger.

IMPORTANT!!! The sections below use examples from the `CJEUprocessorDEMO`, which may be downloaded together with this user manual. The demo crawls and processes 625 case law, available online, from the European Court of Justice.

[2]`https://www.mongodb.com`

2 SDFItems

The SDFItems are SDFDependencyTree(s), SDFHead(s), and SDFTag(s). Each of them corresponds to an homonym Java class in the package SDFItems. Those define the input and the output of SDFTagger.

Figure 2 shows the basic architecture of SDFTagger. Starting from a text in natural language (in Figure 2, a fictional text "blah blah blah..."), we first parse the text via a dependency parsing[3], then we transform the result into one or more SDFDependencyTree(s), each of which is defined by a set of SDFHead(s), one for each word in input, connected of one another through grammatical dependency relations.

The SDFDependencyTree(s) are then given in input to an SDFTagger that executes a set of SDFRule(s) on the SDFDependencyTree(s) and returns a set of SDFTag(s).

Figure 2 only depicts the *general* architecture of SDFTagger; the next section shows more specific examples of input and output, serialized on XML files.

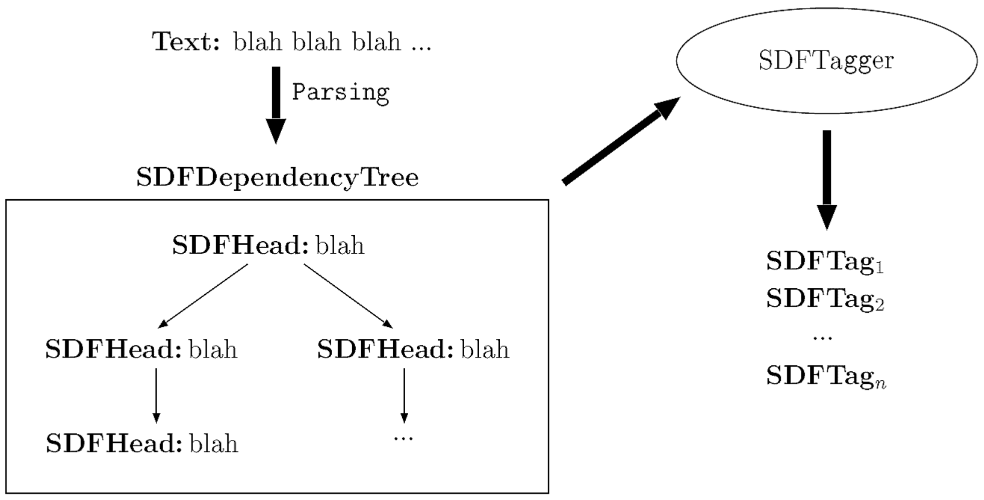


Figure 2: Basic architecture of SDFTagger.

2.1 SDFHead(s)

The object SDFHead is the "atom" of the SDFDependencyTree(s) on which the SDFRule(s) are executed. Each of them refers to a word in the input sentence. SDFHead(s) are connected of one another through grammatical relations such as "subject", "object", "modifier", etc. as it is standard in dependency grammar theories. Dependency relations are oriented: if there is dependency relation from SDFHead$_A$ to SDFHead$_B$, then SDFHead$_B$ is the *governor* of SDFHead$_A$, while SDFHead$_A$ is one of the *dependents* of

[3]Any dependency parser is accepted, i.e., SDFTagger technology is parser-neutral. The `CJEUprocessorDEMO` uses the Stanford CoreNLP (`https://stanfordnlp.github.io/CoreNLP`), version 3.8.0, whose jar classes are downloaded together with the demo.

SDFHead$_B$. Each SDFHead can have at most a governor while it can have more than one dependents (or zero; in that case the SDFHead is a leaf of the tree).

The SDFDependencyTree objects are just clusters of SDFHead(s) where we identify a single SDFHead having no governor; this is the *root* of the SDFDependencyTree.

All methods in SDFHead and SDFDependencyTree only allow to get the values of the attributes of the two objects. Attributes in SDFHead correspond to morphological feature of the referred word. SDFHead defines a set of mandatory features, while it also allows to add (any number of) optional features. Mandatory features are:

- `Form`: the word in input, exactly as it appears in the input sentence.
- `Lemma`: the canonical version of the `Form`, i.e., the `Form` without the inflections. For instance, "eat" is the `Lemma` of the `Form` "ate", which is the verb "eat" in past simple tense.
- `POS`: the part of speech (e.g., "noun", "verb", "adjective", etc.).
- `endOfSentence`: a boolean feature (i.e., its value could be either "true" or "false") to mark whether the word is the last one of the sentence where it occurs.
- `Governor`: the SDFHead governing this one.
- `Label`: the label on the dependency relation to the governor (e.g., "subject", "object", "modifier", etc.).
- `dependents`: the SDFHead(s) depending on this one; of course, all these SDFHead(s) have this one as `Governor`.

An unlimited number of optional features may be then added to an SDFHead; they are stored within a Java object of type Hashtable<String,String>. For instance, we can have an optional feature "Gender" for SDFHead(s) that are nouns, pronouns, adjectives, and participle verbs; possible values for "Gender" could be "M", "F" and "N", corresponding to the masculine, feminine, or neutral gender. Or, we could introduce an optional feature "Type" for adjectives, in order to distinguish between qualifier adjectives, possessive adjectives, cardinal adjectives, etc.

Features are generally provided by the dependency parser used to parse the input text. It is assumed that mandatory features are those that all dependency parsers are able to provide. Optional features are all other features that can be assigned to a word.

Note that it is also possible to add optional features that do not come from the dependency parser used to parse the text. As an example, consider the `CJEUprocessorDEMO`, which processes case law from the European Court of Justice, available online in HMTL format. The HTML files mark certain information in bold or with a special class "title"; bold is in particular used to mark the parties involved in the trial. Of course, the information conveyed by this special formatting is highly relevant for processing the case law, thus it must be preserved in the input SDFHead(s). To this end, the demo introduces an optional feature "Font" whose possible values are "Title", "Bold", and "Normal". All SDFHead(s) have this feature; those within a title or in bold are associated with the first and the second value respectively, while all other words are associated with the value "Normal". SDFTagger can then define SDFRule(s) that search for these values and associate special SDFTag(s) with the SDFHead(s) in bold or within a title.

2.2 SDFTag(s)

The SDFTag(s) objects are assigned by the SDFRule(s) executed by an instance of SDF-Tagger. They are given in output by the main method `tagTrees` of the SDFTagger Java class. Then, further actions may be taken to post-process the output SDFTag(s); for instance, the utilities illustrated in the next section allow to build an annotated XML file starting from another XML file and a set of SDFTag(s). The Java object SDFTag includes the following public attributes:

- `tag`: contains the string associated with the `taggedHead`.
- `taggedHead`: this is the SDFHead to which the tag is assigned.
- `priority`: this is the priority of the SDFRule that assigned `tag` to `taggedHead`; SDFTagger returns the set of all SDFTag(s) assigned by its SDFRule(s) ordered on their priority (from the highest priority to the lowest).
- `idSDFRule`: the id of the SDFRule that assigned `tag` to `taggedHead`.
- `idInstance`: since the same SDFRule could assign tags on the same SDFHead(s) in multiple instantiations, we also associate with the SDFTag an id of the instantiation (which is an incremental number assigned by SDFTagger). Everytime an SDFRule is executed, it is assigned a new `idInstance`, which is then written in the SDFTag(s) assigned by the SDFRule. All SDFTag(s) assigned by (an instance of an) SDFRule may be found by grouping them on the `idInstance` attribute.

3 XMLUtilities

This package includes two classes, each of which having a single public method with the same name of the class.

3.1 XML2SDFDependencyTrees

This class has a single (homonym) public method:

public static *ArrayList<SDFDependencyTree>*
XML2SDFDependencyTrees(*Element DependencyTrees*)

The input parameter is a `org.jdom.Element`[4], in the following format:

```
<DependencyTrees>
    <DependencyTree>
    <line eId="1">
      <Form>For</Form>
      <Lemma>for</Lemma>
      <POS>IN</POS>
      <Governor>3</Governor>
      <Label>case</Label>
    </line>
    <line eId="2">
      <Form>the</Form>
      <Lemma>the</Lemma>
      <POS>DT</POS>
      <Governor>3</Governor>
      <Label>det</Label>
    </line>
    <line eId="3">
      <Form>purposes</Form>
      <Lemma>purpose</Lemma>
      <POS>NNS</POS>
      <Governor>15</Governor>
      <Label>nmod</Label>
    </line>
    ...
  </DependencyTree>
</DependencyTrees>
```

Each `<DependencyTrees>` contains one or more `<DependencyTree>`(s) which contain one or more `<line>`(s). Each `<line>` corresponds to a node of the dependency tree and it will be associated with an SDFHead by XML2SDFDependencyTrees.

[4]`http://www.jdom.org`

Each `<line>` is associated with an `eId`; this is an integer used as pointer for the dependency syntactic relations. The `eId` is unique within the `<DependencyTree>`. `eId`(s) must be incremental integers starting from 1. Each `<line>` has five *mandatory* sub-Element(s): `<Form>`, `<Lemma>`, `<POS>`, `<Governor>`, and `<Label>`, which correspond to the well-known morpho-syntactic features that may be beared by the nodes of a dependency tree. `<Governor>` specifies the `eId` of the node, within the same dependency tree, which governs the one associated with the line; in other words, the corresponding dependency relation start from this `<line>`, ends to the one indexed by `eId`, and it has the label specified in `<Label>`.

Within each `<DependencyTree>`, there is one (and only one) `<line>` having, among its sub-Element(s), `<Governor>0</Governor>` and `<Label>ROOT</Label>`. This `<line>` corresponds to the root of the dependency tree.

The five mandatory sub-Element(s) of `<line>` are given by an (external) dependency parser. The ones shown above are obtained via the Stanford CoreNLP[5].

Figure 3 below shows another example: the dependency tree of the sentence "I ate an apple." obtained via the Stanford CoreNLP; the graphical representation of the dependency tree is shown on the right.

As said above, `<Form>`, `<Lemma>`, `<POS>`, `<Governor>`, and `<Label>` are *mandatory* sub-Element(s) of `<line>`. It is then possible to add further optional sub-Element(s), referring to other morpho-syntactic features. An examples is shown in the next subsection; more examples are then shown in section 2 and subsection 4.7 below.

The method XML2SDFDependencyTrees takes in input a `<DependencyTrees>` and returns an ArrayList of instances of the Java class SDFDependencyTree, one for each `<DependencyTree>`.

The original input XML file may contain other XML tags. The idea is the one of parsing the content of each instance of org.jdom.Text via a dependency parser, e.g., the Stanford CoreNLP, transforming the result of the dependency parser into an org.jdom.Element `<DependencyTrees>`, substitute the org.jdom.Text with the `<DependencyTrees>`, and storing the result into a new XML file that may be later processed via the method method XML2SDFDependencyTrees.

The ArrayList<SDFDependencyTree> returned by XML2SDFDependencyTrees is then given in input to an instance of SDFTagger that executes certain SDFRule(s) to tag some of the nodes in the SDFDependencyTree(s); a set of (prioritized) SDFTag(s) is returned by SDFTagger.

Finally, the Java class SDFDependencyTrees2XML, described in the next subsection, implements the reverse transformation by adding new XML tags, on the basis of the SDFTag(s) returned by SDFTagger. The XML tags in the original XML input file are preserved. An example is shown in the next subsection.

[5] `https://stanfordnlp.github.io/CoreNLP`

```
<DependencyTree>
  <line eId="1">
    <Form>I</Form>
    <Lemma>I</Lemma>
    <POS>PRP</POS>
    <Governor>2</Governor>
    <Label>nsubj</Label>
  </line>
  <line eId="2">
    <Form>ate</Form>
    <Lemma>eat</Lemma>
    <POS>VBD</POS>
    <Governor>0</Governor>
    <Label>ROOT</Label>
  </line>
  <line eId="3">
    <Form>an</Form>
    <Lemma>a</Lemma>
    <POS>DT</POS>
    <Governor>4</Governor>
    <Label>det</Label>
  </line>
  <line eId="4">
    <Form>apple</Form>
    <Lemma>apple</Lemma>
    <POS>NN</POS>
    <Governor>2</Governor>
    <Label>dobj</Label>
  </line>
  <line eId="5">
    <Form>.</Form>
    <Lemma>.</Lemma>
    <POS>Punctuation</POS>
    <Governor>2</Governor>
    <Label>punct</Label>
  </line>
</DependencyTree>
```

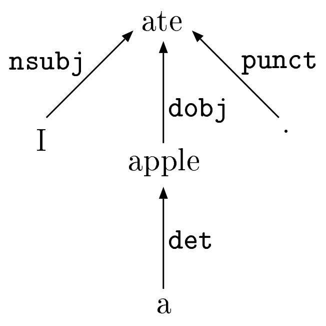


Figure 3: The analysis of the sentence “I ate an apple.” via the Stanford CoreNLP.

3.2 SDFDependencyTrees2XML

This class has a single (homonym) public method:

> public static *ArrayList<Content>*
> **SDFDependencyTrees2XML**(*ArrayList<SDFDependencyTree> trees, Hashtable<SDFHead, SDFTag> taggedSDFHeads*)

The method takes in input the sequence of SDFDependencyTree(s), associated with an XML org.jdom.Element `<DependencyTrees>`, and an Hashtable associating each SDFHead belonging to these SDFDependencyTree(s) with an SDFTag. The method outputs an ArrayList of org.jdom.Content(s); however, indeed this ArrayList only contains instances of org.jdom.Text and org.jdom.Element (org.jdom.Content is the direct superclass of both org.jdom.Text and org.jdom.Element).

The ArrayList of Text(s) and Element(s) is intended to substitute, within a new output XML file, the original Text corresponding to the ArrayList<SDFDependencyTree> in input. The Element(s) in the output ArrayList are obtained out of the SDFTag(s): if an SDFHead is associated with an SDFTag, a corresponding Element enclosing that SDFHead is created in the output file; in case two consecutive SDFHead(s) are associated with the same SDFTag, they are both enclosed within the same corresponding Element.

For instance, consider the following (input) XML file:

```
<?xml version="1.0" encoding="utf-8"?>
<document xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <sentence id="1">I ate an apple.</sentence>
</document>
```

By using the Stanford CoreNLP, we create following XML file, where the underspecified Element `<DependencyTree>...</DependencyTree>` is the `<DependencyTree>` shown in Figure 3 above.

```
<?xml version="1.0" encoding="utf-8"?>
<document xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <sentence id="1">
    <DependencyTrees>
      <DependencyTree>...</DependencyTree>
    </DependencyTrees>
  </sentence>
</document>
```

The `<DependencyTrees>` is given in input to the method XML2SDFDependencyTrees described in the previous section; this transforms the XML file into an ArrayList of SDFDependencyTree(s), which is in turn given in input to an instance of SDFTagger, thus obtaining an ArrayList of SDFTag(s).

The ArrayList<SDFTag> is transformed into a Hashtable<SDFHead, SDFTag> via post-processing operations. In fact, it is possible that the same SDFHead is associated with different tags within the ArrayList<SDFTag>; on the other hand, the method

SDFDependencyTrees2XML requires each SDFHead to be associated with at most one SDFTag, thus the need of an Hashtable as parameter of the method. As explained above, each SDFTag features a priority, which can be used for choosing the SDFTag to be associated with an SDFHead, in case of conflicts.

Suppose SDFTagger associates the SDFHead corresponding to the word "I" with the SDFTag "subject" and the SDFHead(s) corresponding to the words "an" and "apple" with the SDFTag "object". In other words, suppose that the Hashtable "taggedSDF-Heads", the second parameter of the method SDFDependencyTrees2XML, specifies the following associations:

```
taggedSDFHeads =
{
    "I" -> "subject",
    "an" -> "object",
    "apple" -> "object"
}
```

Then, the following XML output file is eventually obtained:

```
<?xml version="1.0" encoding="utf-8"?>
<document xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <sentence id="1">
        <subject>I</subject>
        ate
        <object>an apple</object>
        .
    </sentence>
</document>
```

Indeed, in order to generate that XML file, another *optional* sub-Element of the XML tag `<line>` needs to be specified: `<blanksBefore>`. This specifies the number of blanks that need to preceed each word. `<blanksBefore>` is a "command" needed by SDFDependencyTrees2XML to avoid a space between the dot and the word "apple", i.e., to generate "I ate an apple." in place of " I ate an apple .".

If `<blanksBefore>` is not specified, SDFDependencyTrees2XML generates the latter, in that it considers '1' as default value. In the example above, all `<line>`(s) but the ones associated to "I" and "." specify 1 as `<blanksBefore>`; "I" and "." specify '0'. Therefore, the input `<line>`(s) are the following:

```
<line eId="1">
  <Form>I</Form>
  <Lemma>I</Lemma>
  <POS>PRP</POS>
  <blanksBefore>0</blanksBefore>
  <Governor>2</Governor>
  <Label>nsubj</Label>
</line>
```

```
<line eId="2">
  <Form>ate</For
  <Lemma>eat</Lemma>
  <POS>VBD</POS>
  <blanksBefore>1</blanksBefore>
  <Governor>0</Governor>
  <Label>ROOT</Label>
</line>
```

```
<line eId="3">
  <Form>an</Form>
  <Lemma>a</Lemma>
  <POS>DT</POS>
  <blanksBefore>1</blanksBefore>
  <Governor>4</Governor>
  <Label>det</Label>
</line>
```

```
<line eId="4">
  <Form>apple</Form>
  <Lemma>apple</Lemma>
  <POS>NN</POS>
  <blanksBefore>1</blanksBefore>
  <Governor>2</Governor>
  <Label>dobj</Label>
</line>
```

```
<line eId="5">
  <Form>.</Form>
  <Lemma>.</Lemma>
  <POS>Punctuation</POS>
  <blanksBefore>0</blanksBefore>
  <Governor>2</Governor>
  <Label>punct</Label>
</line>
```

4 SDFTagger - core classes

This section illustrates the core classes of SDFTagger, i.e., the ones that the users must instantiate and extend in order to use the SDFTagger rule-based system. The SDFRule(s) to feed SDFTagger with are codified in XML format, and can be either manually edited or automatically generated via scripts or statistical NLP procedures.

4.1 SDFTagger and SDFTaggerConfig

`SDFTagger.java` is of course the main class of the SDFTagger system. Its constructor requires as input parameter an instance of the class `SDFTaggerConfig.java`, which clusters all parameters needed by SDFTagger to associated the SDFHead(s) in a set of SDFDependencyTree(s) with SDFTag(s).

Usually, *more than one* instances of SDFTagger is required, each of which identifies a restricted set of linguistic patterns. For instance, as it will be explained below in section 6, the `CJEUprocessorDEMO` instantiates three SDFTagger(s), each of which on a different *subclass* of SDFTaggerConfig:

- **SDFTaggerConfigForStructuralTagging**, needed to identify the structure of the case law, i.e., sections, subsections, paragraphs, titles, itemizations, etc.;

- **SDFTaggerConfigForPartyClassificationOnKeywords**, needed to classify the parties of a case law as either company or institution, once the words in the party's proper name have been identified;

- **SDFTaggerConfigForEntityLinking**, needed to identify the parties classified by SDFTaggerConfigForPartyClassificationOnKeywords in the rest of the text, either through their standard name or through a variant of their standard name.

In other words, these three classes extend the basic class SDFTaggerConfig and specify three different clusters of SDFTagger's parameters, which are:

- The instance of **SDFNodeConstraints**, needed to process the optional features of the SDFHead, if any (see subsection 4.7 below).

- The instance of **SDFLogger**, needed to trace the execution of the SDFRule(s) for debugging purposes (see section 4.8). The XML file where SDFLogger will store the logging of the SDFTagger execution is given as parameter of SDFTaggerConfig; it is then passed to SDFLogger when its instance is created.

- The instance of **KBInterface** and the **local paths** of the SDFRule(s) that SDFTagger must load and execute (see section 5 below).

Once an object of SDFTagger is instantiated via its constructor, the only public method that can be called is:

public *ArrayList<SDFTag>* **tagTrees**(*ArrayList<SDFDependencyTree> trees*)

The method `tagTrees` associates (and returns) a set of SDFTag(s) with the SDFHead(s) of the SDFDependencyTree(s) in input.

Besides SDFTagger, the other two principal core classes are SDFNode and SDFRule, which will be described in the two next subsections respectively. Although these classes must not be extended by the user, their descriptions will clarify how the SDFTagger system works and how it ought to be used.

4.2 SDFNode(s)

The SDFRule(s) of an SDFTagger are executed on a chain of SDFNode(s). SDFNode(s) are *wrappers* of the SDFHead(s) in the input SDFDependencyTree(s). Figure 5 shows the chain of SDFNode(s) corresponding to the SDFDependencyTree obtained from the XML representation in Figure 3.

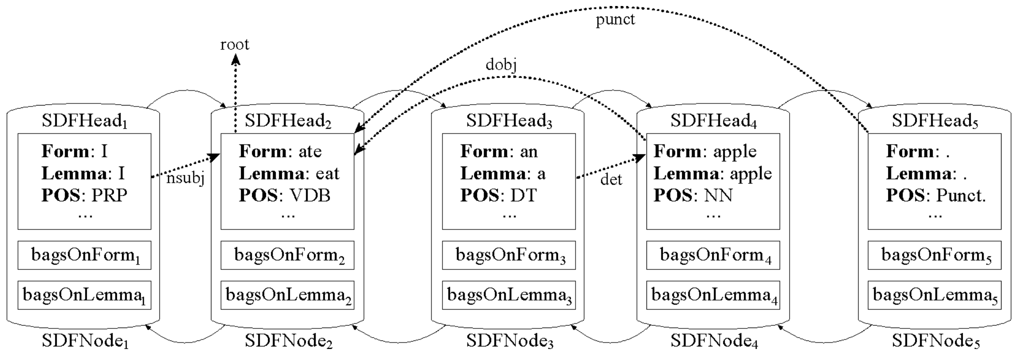


Figure 4: The SDFNode(s) bidirectional chain for the sentence “I ate an apple.”

The chain is *bidirectional*, i.e., it can be crossed in both directions: from the left to the right and from the right to the left. An SDFNode contains an SDFHead plus the bags defined on the form and on the lemma of the SDFHead, which will be explained and described below. As said above, the SDFHead(s) are further connected among them via dependency relations (shown via dotted arrows in Figure 5).

The representation in Figure 5 allows SDFRule(s) to “move” into four possible directions: left, right, up, and down. Specifically, the left and right directions allow to pose conditions on the SDFHead(s) along their surface order, while the up and down directions allow to pose conditions on the SDFHead(s) along their dependency relations (governors or dependents).

It is important to understand that SDFTagger builds and uses a *single* bidirectional chain of SDFNode(s) for *all* SDFDependencyTree(s) in input. In other words: there is *not* a different chain of SDFNode(s) for each dependency tree in input. Of course, this entails that the SDFHead(s) within a chain of SDFNode(s) are not all connected via dependency relations; rather, the set of SDFHead(s) belonging to a chain of SDFNode(s) can be *partitioned* on the SDFDependencyTree(s).

The SDFHead(s) that end an SDFDependencyTree have the `endOfSentence` feature set to “true”, while all other SDFHead(s) have the `endOfSentence` feature set to “false”. For instance, in Figure 5, the rightmost SDFNode, wrapping the SDFHead corresponding to the dot “.”, has the `endOfSentence` feature set to “true”, while all other SDFNode(s), wrapping the SDFHead(s) corresponding to the words “I”, “ate”, “an”, and “apple”, have the `endOfSentence` feature set to “false”.

Finally, bagsOnForms and bagsOnLemma specify the *bags of words* where the SDFHead occurs. SDFTaggerConfig of SDFTagger includes two special attributes “rootDirectoryBags” and “localPathsBags” that specify the local paths of the bags of words used by SDFTagger. For instance, the `CJEUprocessorDEMO` uses the following bag of forms to store the list of letters of English alphabeth:

```
<Bag type="Form" name="Letters">
   <instance>a</instance>
   <instance>b</instance>
   <instance>c</instance>
   <instance>d</instance>
   <instance>e</instance>
            ...
   <instance>x</instance>
   <instance>y</instance>
   <instance>z</instance>
</Bag>
```

Letters are needed by the `CJEUprocessorDEMO` to recognize, in the case law of the European Court of Justice, enumerations in the form:

a. First item.

b. Second item.

   (a) First subitem.
   (b) Second subitem.

c. Third item.

d. Fourth item.

The next section will explain how the SDFRule(s) are able to pose conditions on the SDFHead(s). Bags of words allow to compactly pose conditions in the form:

```
<head>
   <Bag name="Letters" />
</head>
```

rather than (equivalent) multiple disjunctive conditions in the form:

```
<head>
   <Form>a</Form>
</head>
<head>
   <Form>b</Form>
</head>
<head>
   <Form>c</Form>
</head>
...
<head>
   <Form>z</Form>
</head>
```

Therefore, bags of words allow to cluster multiple Form(s) or Lemma(s)[6] in order to: (1) write more compact conditions (2) in case the bag is used in multiple SDFRule(s), avoid multiple copies of the same (long) condition, i.e., updating the bag in a single location.

The names of all bags defined on an SDFHead are loaded, at the beginning of the processing, within the SDFNode that wraps that SDFHead, so that the (possible, potential) conditions of the SDFHead on its bags of words can be quickly checked.

4.3 SDFRule(s)

Each instance of the class `SDFRule.java` corresponds to an if-then rule that executes on the chain of SDFNode(s). If all the conditions in the SDFRule are satisfied, the SDFNode(s) that satisfy them are tagged with the SDFTag(s) specified in the SDFRule.

SDFRule(s) are written in XML; examples are available in the `CJEUprocessorDEMO` or below. However, XML cannot be executed directly: SDFRule(s) need to be compiled in a special bytecode. The bytecodes, which consume less memory, are loaded and executed.

SDFTaggerConfig specifies the paths of both the XML files and the corresponding compiled files (having extension ".sdf").

The XMLFilesManager class, illustrated below in section 5, checks if the paths of the XML files and the corresponding bytecodes are aligned, every time a new SDFTagger is instantiated. If the files are misaligned, either because new SDFRule(s) in XML have been added or because the previous ones have been deleted or modified, the XMLFilesManager automatically recompiles the XML files into the bytecodes. Therefore, the user does not need to worry and manually check if the bytecodes are up-to-date: every update is handled automatically by XMLFilesManager.

On the other hand, as it will be explained below in section 5, the class MongoDB-Manager loads the bytecodes in MongoDB. While using MongoDB, the system does not automatically check if the XML are changed: in such a case, the user needs to manually check by himself/herself, and reload the MongoDB to include the latest updates.

[6]Depending on the parameter `type` of the `<Bag>`; in the example above, we have `type="Form"`; other bags, which cluster Lemma(s), have `type="Lemma"`.

The XML format of the SDFRule(s) is well defined. Only well-formed SDFRule(s), as defined below in subsection 4.6, are compiled. If the XML file is ill-formed, an exception is thrown when we try to compile it. The message of the exception contains a pointer to a specific paragraph of subsection 4.6, where more details are available.

XML files can be manually written or automatically generated via statistical NLP (hybrid approach). SDFTagger loads all SDFRule(s) specified in its SDFTaggerConfig and executes them on the chain of SDFNode(s). The set of SDFTag(s) assigned by the SDFRule(s) that succeeded is returned, ordered on the priority of the SDFTag(s).

4.4 The architecture of the SDFRule(s)

SDFRule(s) are specified via the XML tag `SDFRule`. This tag has two attributes: `id` and `priority`, whose values are reported in the SDFTag(s) assigned by the SDFRule, in case the SDFRule succeeds. Within `SDFRule`, it is then possible to specify seven different subconditions:

- `headAlternatives`, which includes one or more `head`.
- `prevStarAlternatives`, which includes one or more `prev` (or `notPrev`).
- `prevAlternatives`, which includes one or more `prev` (or `notPrev`).
- `nextStarAlternatives`, which includes one or more `next` (or `notNext`).
- `nextAlternatives`, which includes one or more `next` (or `notNext`).
- `governorAlternatives`, which includes one or more `governor` (or `notGovernor`).
- `dependentsAlternatives`, which includes one or more `dependents`

Only `headAlternatives` is mandatory. All other subconditions are optional. Each `*Alternatives` represents a *disjunction*. Each of them includes *one or more* XML tags and it is satisfied if and only if *at least one* of these tags is satisfied. For instance, an `headAlternatives` is satisfied is at least one of its `head` is satisfied, each `nextAlternatives` is satisfied if at least one of its `next` is satisfied, etc.

Subconditions are *recursively* asserted within each XML element `<prev>...</prev>`, `<next>...</next>`, `<governor>...</governor>`, and `<dependent>...</dependent>`: those elements must contain one `headAlternatives` and, possibly, some or all the other six (optional) subconditions, in a recursive fashion.

`<governor>...</governor>` and `<dependent>...</dependent>` can also contain a special condition `<labelAlternatives>...</labelAlternatives>`, explained below.

```
<SDFRule id="x" priority="y">
    <headAlternatives>
         <head>...</head>
         ...
         <head>...</head>
    </headAlternatives>
    <prevStarAlternatives>
         <prev maxDistance="ps1">...</prev>
         ...
         <prev maxDistance="psn">...</prev>
    </prevStarAlternatives>
    <prevAlternatives>
         <prev maxDistance="p1">...</prev>
         ...
         <prev maxDistance="pn">...</prev>
    </prevAlternatives>
    <nextStarAlternatives>
         <next maxDistance="ns1">...</next>
         ...
         <next maxDistance="nsn">...</next>
    </nextStarAlternatives>
    <nextAlternatives>
         <next maxDistance="n1">...</next>
         ...
         <next maxDistance="nn">...</next>
    </nextAlternatives>
    <governorAlternatives>
         <governor maxHeight="g1">...</governor>
         ...
         <governor maxHeight="gn">...</governor>
    </governorAlternatives>
    <dependentsAlternatives>
         <dependents>
              <dependent maxDepth="d11">...</dependent>
              ...
              <dependent maxDepth="d1n">...</dependent>
         </dependents>
         ...
         <dependents>
              <dependent maxDepth="dm1">...</dependent>
              ...
              <dependent maxDepth="dmn">...</dependent>
         </dependents>
    </dependentsAlternatives>
</SDFRule>
```

4.4.1 `<headAlternatives>` and `<head>`

The tag `<headAlternatives>` is used to create clusters of (disjunctive) `<head>`(s): an `<headAlternatives>` is satified iff *at least one* of its `<head>`(s) is satisfied. The tag `<head>` is used to pose conditions on a single SDFNode. The are several kinds of conditions that can be specified on a `<head>`:

- `Form`, `Lemma`, and `POS`: they are satisfied iff the string matches the corresponding one defined on the SDFNode.
- `notForm`, `notLemma`, `notPOS`: they are satisfied iff the string specified in the conditions *do not*[7] match the corresponding one on the SDFNode.
- `endOfSentence`: if its value is "true", the condition is satisfied iff the SDFNode ends the sentence, i.e., iff it is the rightmost one in the SDFDependencyTree.
- Optional features, such as `<blanksBefore>` in subsection 3.2 above. Their satisfaction can be defined ad-hoc as explained below in subsection 4.7.

As an example, consider the following `<head>...</head>`, which is satisfied by all words which are verbs with lemma "eat" (i.e., "eat", "eats", "ate", etc.) that do not end a sentence, i.e., that are not the rightmost SDFHead of a dependency tree.

```
<head>
    <Lemma>eat</Lemma>
    <POS>VBD</POS>
    <endOfSentence>false</endOfSentence>
</head>
```

On the other hand, the following `<head>...</head>` is satisfied only by the word "ate"; note that, in this case, `<Lemma>eat</Lemma>` and `<POS>VBD</POS>` are indeed redundant as these are the only available lemma and part of speech for the word "ate".

```
<head>
    <Form>ate</Form>
    <Lemma>eat</Lemma>
    <POS>VBD</POS>
    <endOfSentence>false</endOfSentence>
</head>
```

[7]Note that it is possible to have multiple `not*` conditions. For instance, the following `head`:

```
<head>
  <notLemma>eat</notLemma>
  <notLemma>run</notLemma>
  <notLemma>see</notLemma>
  <POS>VBD</POS>
</head>
```

matches all verbs which are not inflections of the verb "to eat", nor of the verb "to run", nor of "to see".

It is also possible to specify empty `<head>`(s), i.e.:

```
<head>
</head>
```

This `<head>` is satisfied by *any* SDFHead. Specifying empty `<head>`(s) allows, for instance, to recognize words having at least one dependent or words having the governor (i.e., words that are not roots). Subsection 4.6 below shows examples of conditions employing empty `<head>`(s).

The words that matches the `<headAlternatives>` associated with an `<SDFRule>` are said the *trigger words* for that SDFRule, i.e., the words from which an instance of the SDFRule is executed. For instance, the trigger words of the following `<SDFRule>` are all inflections of the verb "to eat", i.e., "eat", "eats", "ate", etc.

```
<SDFRule id="1" priority="10">
    <tag>inflection of eat</tag>
    <headAlternatives>
         <head>
              <Lemma>eat</Lemma>
         </head>
    </headAlternatives>
</SDFRule>
```

This SDFRule is triggered on each inflection of the verb "to eat" and, to each of them, it assigns the tag "inflection of eat".

4.4.2 `<prevAlternative>`, `<prev>`, `<nextAlternative>`, and `<next>`

`<prevAlternative>` and `<nextAlternative>` allow to pose (sub)conditions in the left and right directions respectively, i.e., they allow to pose (sub)conditions on the precedent SDFNode(s) and the subsequent SDFNode(s) respectively.

Their inner XML items are, respectively, `<prev>` and `<next>`, or the negative counterparts `<notPrev>` and `<notNext>` that are explained below in subsection 4.4.6.

The structure of `<prev>` and `<next>` is the same of `<SDFRule>`, i.e., they obligatorily include an `<headAlternative>` and they can recursively include one or more of the seven subconditions at the beginning of subsection 4.4. `<prevAlternative>`(s) (and `<nextAlternative>`(s)) are satisfied if at least one of their `<prev>`(s) (or `<next>`)(s) are.

`<prev>` and `<next>` have the mandatory attribute `maxDistance`, which is an integer equal or greater than 1. `maxDistance` specifies the maximum number of SDFNode(s) that are considered in the left (`<prev>`) or right (`<next>`) direction.

For instance, consider the following chain of SDFNode(s) and the SDFRule below it, which is triggered on the word "the".

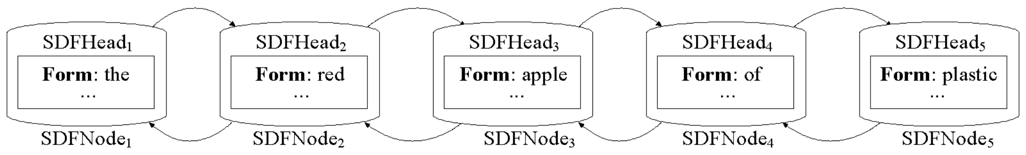


Figure 5: The SDFNode(s) bidirectional chain for the phrase "The red apple of plastic"

```
<SDFRule id="1" priority="10">
    <headAlternatives>
        <head>
            <Form>the</Form>
        </head>
    </headAlternatives>
    <nextAlternatives>
        <next maxDistance="1">
            <tag>fruit</tag>
            <headAlternatives>
                <head>
                    <Form>apple</Form>
                </head>
            </headAlternatives>
        </next>
        <next maxDistance="2">
            <tag>fake fruit</tag>
            <headAlternatives>
                <head>
                    <Form>apple</Form>
                </head>
            </headAlternatives>
            <nextAlternatives>
                <next maxDistance="2">
                    <headAlternatives>
                        <head>
                            <Form>plastic</Form>
                        </head>
                    </headAlternatives>
                </next>
            </nextAlternatives>
        </next>
    </nextAlternatives>
</SDFRule>
```

The `<nextAlternative>` of the `<SDFRule>` includes two `<next>`(s), the former having `<maxDistance>` equal to 1, the latter having `<maxDistance>` equal to 2.

The first `<next>` fails[8], in that it is executed only on the *immediately* subsequent word, i.e., "red". This does not satisfy the `<headAlternatives>` of the `<next>`.

On the other hand, the second `<next>` succeeds: since `<maxDistance>` is equal to 2, the two subsequent words of "the", i.e., "red" and "apple", are considered. The `<headAlternatives>` of the `<next>` fails on "red", but it succeeds on "apple"; note the second `<next>` includes a sub-`<nextAlternative>` that requires the existence of the word "plastic" after at most two words from the one that satisfies its `<headAlternatives>`, i.e., "apple". The word "plastic" is indeed found after two words from "apple", so that the sub-`<nextAlternative>` is satisfied, and so the `<next>`, and so the `<nextAlternative>` of the `<SDFRule>`, and so the overall `<SDFRule>`.

Therefore, the SDFRule tag the word "apple" with the string "fake fruit"; on the other hand, note that, if the first `<next>` would have succeeded, "apple" would have been tagged with the string "fruit".

4.4.3 `<prevStartAlternative>` and `<nextStartAlternative>`

`<nextStarAlternatives>` and `<prevStarAlternatives>` behave as their non-`star` versions, but the inner `<next>`(s) or `<prev>`(s) may be executed from zero to n times, every time restarting from the word reached during the previous execution of the `<next>`(s) or `<prev>`(s). In other words, `<nextStarAlternatives>` and `<prevStarAlternatives>` are intended as the application of the Kleene star operator[9] to `<nextAlternatives>` and `<prevAlternatives>`. Note that the two conditions `<nextStarAlternatives>` and `<prevStarAlternatives>` are not intended to replace `<nextAlternatives>` and `<prevAlternatives>`. In other words, they can (and should) be used *together*.

As an example, consider the `SDFRule` below, including both a `<nextAlternatives>` and a `<nextStarAlternatives>` subconditions[10]. The `SDFRule` is successful, and it assigns the tag "fruit" to the word "apple", in all the following phrases:

- "*The apple*": in such a case, there is not any adjective between "the" and "apple". Therefore, the `<nextStarAlternatives>` subcondition is simply ignored. As said above, `<nextStarAlternatives>` and `<prevStarAlternatives>` parallel the Kleene star, so that they can be satisfied zero or more times.

- "*The red apple*": in such a case, an adjective (i.e., a word having POS equal to "JJ", when the text is parsed via the Stanford CoreNLP), is encountered between "the" and "apple". The `<next>` within the `<nextStarAlternatives>` is satisfied by the adjective so that the SDFRule is able to "jump over" it and the `<nextAlternatives>` can be executed starting from the word "apple".

[8]The `<next>`(s) are executed in the same order they appear in the `<SDFRule>`. The first that succeeds satisfies the `<nextAlternatives>` where it occurs. Other `<next>`(s), subsequent to the (first) one that succeeded, are ignored, even if they would have also succeeded. The same hold for `<prev>`(s), `<governor>`(s), and `<dependent>`(s).

[9]`https://en.wikipedia.org/wiki/Kleene_star`

[10]In the `<nextStarAlternatives>` subcondition of the SDFRule, "JJ" is the part-of-speech assigned by the Stanford CoreNLP to adjectives; see `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`

- “*The red big apple*”: in such a case, the `<nextStarAlternatives>` is first executed on “red”, which satisfies the inner `<next>`. `<nextStarAlternatives>` is then executed again on the subsequent word, which is again an adjective, so that the inner `<next>` is again satisfied. `<nextStarAlternatives>` is then executed again on the word “apple”, but this time it fails, so that the execution continues from the `<nextAlternatives>`, starting from the word “apple”. As in the previous cases, the `<nextAlternatives>` is satisfied by the word “apple”, so that the whole `<SDFRule>` succeeds, and “apple” is associated with the tag “fruit”.

- “*The red big odd apple*”: same as the previous case, with the difference that `<nextStarAlternatives>` is executed three times on the three adjectives, before failing on the word “apple”.

```
<SDFRule id="1" priority="10">
    <headAlternatives>
         <head>
              <Form>the</Form>
         </head>
    </headAlternatives>
    <nextStarAlternatives>
         <next maxDistance="1">
              <headAlternatives>
                   <head>
                        <POS>JJ</POS>
                   </head>
              </headAlternatives>
         </next>
    </nextStarAlternatives>
    <nextAlternatives>
         <next maxDistance="1">
              <tag>fruit</tag>
              <headAlternatives>
                   <head>
                        <Form>apple</Form>
                   </head>
              </headAlternatives>
         </next>
    </nextAlternatives>
</SDFRule>
```

4.4.4 `<governorAlternatives>`, `<governor>`, `<labelAlternatives>`, and `<label>`

`<governorAlternatives>` and `<governor>` behave exacly as `<prevAlternatives>` and `<prev>`, or as `<nextAlternatives>` and `<next>`, but on the “up” direction, i.e., along the grammatical governing relations of the words.

Since grammatical (dependency) relations are also associated with a label, e.g., "subject", "object", etc., each `<governor>` may also include a `<labelAlternatives>` condition, which is satisfied iff at least one of the `label`(s) specified therein corresponds to the label of the dependency relation. `<governor>` also includes a mandatory attribute `maxHeight` that specifies the number of governor(s) that we have to consider. For instance, in case `maxHeight=2`, the `<governor>` condition considers the governor of the word and, in case it fails on the latter, also the governor of the governor. The label on which SDFTagger enforces the `<labelAlternatives>` condition is always the one *immediately* below the SDFNode on which the `<governor>` condition is applied.

As an example, consider again the dependency tree of the sentence "I ate a apple." shown above in Figure 3. In this tree, the word "a", which has been associated by the Stanford CoreNLP with the part of speech "DT", is governed by the noun "apple" and by the verb "ate". The label from "a" to "apple" is "det" while the one from "apple" to "ate" is "dobj". The following SDFRule is triggered on the word "a":

```
<SDFRule id="1" priority="10">
    <headAlternatives>
         <head>
              <POS>DT</POS>
         </head>
    </headAlternatives>
    <governorAlternatives>
         <governor maxHeight="2">
              <labelAlternatives>
                   <label>nsubj<label>
                   <label>dobj<label>
              </labelAlternatives>
              <headAlternatives>
                   <head>
                        <POS>VBD</POS>
                   </head>
              </headAlternatives>
         </governor>
    </governorAlternatives>
    <nextAlternatives>
         <next maxDistance="2">
              <tag>dot</tag>
              <headAlternatives>
                   <head>
                        <Form>.</Form>
                   </head>
              </headAlternatives>
         </next>
    </nextAlternatives>
</SDFRule>
```

The SDFRule checks whether either the governor of “a” or the governor of the governor of “a” is a verb (VBD) connected with the subtree where “a” occurs either with the label “nsubj” or with the label “dobj”. The governor of “a”, i.e., the word “apple”, does not meet this condition. But the word “ate” does, so that the `<governorAlternatives>` condition is satisfied.

Note that the `<governor>` condition does not specify any tag. On the other hand, the SDFRule includes also a `<nextAlternatives>` condition which is satisfied iff one of the two subsequent words of “a” is a dot. Since this condition is also met, the whole SDFRule is satisfied and the dot (“.”) is associated with the tag “dot”.

4.4.5 `<dependentsAlternatives>`, `<dependents>`, and `<dependent>`

The `<dependentsAlternatives>` condition is dual to `<governorAlternatives>`: it is used to pose conditions along the “down” direction, i.e., on the dependents reachable from a given SDFNode. However, there is an important difference: while there could be *at most one* governor of an SDFHead, the latter could have *more than one* dependents.

In order to encompass this distinction, SDFTagger defines two layers: the XML children of `<dependentsAlternatives>`(s) are `<dependents>`(s), each of which clusters together a tuple of one or more `<dependent>`(s). A `<dependentsAlternatives>` condition is satisfied iff *at least one* of its `<dependents>` is satisfied and a `<dependents>` condition is satisfied iff *all* its `<dependent>`(s) are satisfied.

Each `<dependent>` specifies a `maxDepth`, which parallels `maxHeight`, and may likewise contain a `<labelAlternatives>` condition; as in case of `<governorAlternatives>` conditions, the label considered is the one *immediately* starting from the dependent.

Consider again the dependency tree of the sentence “I ate an apple.” shown above in Figure 3 and the SDFRule below, which triggers on the main verb “ate”.

The SDFRule contains a `<dependentsAlternatives>` including two `<dependents>`(s), each of which includes in turn two `<dependent>`(s).

The first `<dependents>` does not satisfy the dependents of “ate”: it requires the existence of a pronoun (PRP) having label “dobj” *and* the word “an” to be immediately connected below the verb. The dependency tree does not include either of these dependents, for the word “ate”, so the `<dependentsAlternatives>` is (doubly) unsatisfied.

On the other hand, the second `<dependents>` does match on the dependency tree, as it requires the existence of a pronoun (PRP) *having label “nsubj”* and the word “an” to be found between two levels of grammatical dependents, i.e., either in the immediate dependents of “ate” or in the dependents of the dependents of “ate”.

The `<dependentsAlternatives>` condition, and so the whole SDFRule, is satisfied. However, note that the SDFRule is indeed ineffective, in that it does not assign any tag to the words it encounters during its evaluation. In other words, the SDFTagger output is always empty for this SDFRule, regardless of either its success or its failure.

```
<SDFRule id="1" priority="10">
    <headAlternatives>
         <head>
              <Lemma>eat</Lemma>
         </head>
    </headAlternatives>
    <dependentsAlternatives>
         <dependents>
              <dependent maxDepth="1">
                   <labelAlternatives>
                        <label>dobj<label>
                   </labelAlternatives>
                   <headAlternatives>
                        <head>
                             <POS>PRP</POS>
                        </head>
                   </headAlternatives>
              </dependent>
              <dependent maxDepth="1">
                   <headAlternatives>
                        <head>
                             <Form>an</Form>
                        </head>
                   </headAlternatives>
              </dependent>
         <dependents>
         <dependents>
              <dependent maxDepth="1">
                   <labelAlternatives>
                        <label>nsubj<label>
                   </labelAlternatives>
                   <headAlternatives>
                        <head>
                             <POS>PRP</POS>
                        </head>
                   </headAlternatives>
              </dependent>
              <dependent maxDepth="2">
                   <headAlternatives>
                        <head>
                             <Form>an</Form>
                        </head>
                   </headAlternatives>
              </dependent>
         <dependents>
    </dependentsAlternatives>
</SDFRule>
```

4.4.6 `<notPrev>`, `<notNext>`, `<notGovernor>`, and `<notDependent>`

`<notPrev>`, `<notNext>`, `<notGovernor>`, and `<notDependent>` are respectively the counterparts of `<prev>`, `<next>`, `<governor>`, and `<dependent>`: they are satisfied iff SDFTagger can *not* identify a previous, subsequent, governing, or depending SDFNode matching the conditions, within at most `maxDistance`, `maxHeight`, or `maxDepth` SDFNodes.

SDFTagger evaluates these conditions as if they were `<prev>`, `<next>`, `<governor>`, and `<dependent>`; if these return true, SDFTagger evaluates them to false and viceversa. It follows that all SDFTag(s) assigned within `<notPrev>`, `<notNext>`, `<notGovernor>`, and `<notDependent>` are ignored: if some SDFHead(s) are associated with SDFTag(s) during the evaluation of these conditions, the latter return false, i.e., they are not satisfied, and therefore the SDFTag(s) have to be discharged.

Note that we can have `<not*>` conditions as alternatives of their positive counterparts. For instance, the following SDFRule:

```
<SDFRule id="1" priority="10">
    <tag>strange apple</tag>
    <headAlternatives>
         <head>
              <Lemma>apple</Lemma>
         </head>
    </headAlternatives>
    <nextAlternatives>
         <notNext maxDistance="3">
              <tag>dot</tag>
              <headAlternatives>
                   <head>
                        <Form>,</Form>
                   </head>
              </headAlternatives>
         </notNext>
         <next maxDistance="1">
              <headAlternatives>
                   <head>
                        <Form>.</Form>
                   </head>
              </headAlternatives>
         </next>
    </nextAlternatives>
</SDFRule>
```

Tags any instance of the words “apple” or “apples” (note, in fact, that the trigger word is defined on the lemma) with the string “strange apple”, iff those words are not followed by a comma within at most three words *or* they are immediately followed by a dot.

On the other hand, the following SDFRule tags with the string “strange apple” every word with lemma “apple” that (1) has the word “red” as dependent but not also the word “three” as dependent, *or* (2) has a determiner as dependent.

```
<SDFRule id="1" priority="10">
    <tag>strange apple</tag>
    <headAlternatives>
         <head>
              <Lemma>apple</Lemma>
         </head>
    </headAlternatives>
    <dependentsAlternatives>
         <dependents>
              <dependent maxDepth="1">
                   <headAlternatives>
                        <head>
                             <Lemma>red</Lemma>
                        </head>
                   </headAlternatives>
              </dependent>
              <notDependent maxDepth="1">
                   <headAlternatives>
                        <head>
                             <Lemma>three</Lemma>
                        </head>
                   </headAlternatives>
              </notDependent>
         <dependents>
         <dependents>
              <dependent maxDepth="1">
                   <headAlternatives>
                        <head>
                             <POS>det</POS>
                        </head>
                   </headAlternatives>
              </dependent>
         <dependents>
    </dependentsAlternatives>
</SDFRule>
```

4.5 Defining priorities on SDFRule(s)

The last component of SDFRule(s) that needs some additional explanations is the attribute `priority` of the `SDFRule` XML tag. As said above, the class SDFTag has also an attribute priority, in which the priority specified on the SDFRule(s) are reported on all output SDFTag(s), provided that the SDFRule succeeds. SDFTagger returns the set of SDFTag(s) ordered on their priority, in order to facilitate post-processing operations.

In other words, SDFTag(s) ought to be considered according to their priorities: the post-processing operations associated with certain SDFTag(s) ought to be executed *unless* there are other higher-priority SDFTag(s) that *overrides* them, either by explicitly blocking them or because they are associated with other post-processing operations that are incompatible with the former.

Priorities are then used in SDFTagger to implement *defeasibility*, which is in turn needed to achieve *hybrid* NLP approaches. The basic idea is to generate an initial knowledge base of SDFRule(s) via statistical methods, while assigning them a certain "low" priority, able to deal with the majority of the cases. Then, higher-priority SDFRule(s) may be manually added in order to override the statistically-generated SDFRule(s) in the exceptional cases that deviates from the standard trend, as long as these are identified.

As an example, consider the SDFRule shown in the next page and taken from the `CJEUprocessorDEMO`. This SDFRule is triggered by any word that is either a number[11] or a letter or a roman number (i.e., "i", "ii", "iii", "iv", "v", etc.). Note that the SDFRule rule uses the feature `<Bag>`, explained at the beginning of this section.

As it should be clear, the SDFRule is used to recognize patterns such as "(a)", "1.", "IV.", etc. These correspond to indexes that are collected by the `CJEUprocessorDEMO` in order to create itemizations in the output XML format. An example where this SDFRule is used, taken from an output file of the `CJEUprocessorDEMO`, is:

```
<blockList>
    <item>
        <num>(a)</num>
        <p>‘‘Union fishing vessel’’ means a fishing vessel...</p>
    </item>
    <item>
         <num>(b)</num>
         <p>‘‘EU waters’’ means waters under the...</p>
    </item>
    <item>
         <num>(c)</num>
         <p>‘‘total allowable catch’’ (TAC) means...</p>
    </item>
    <item>
         <num>(d)</num>
         <p>‘‘quota ’’ means a proportion of the...</p>
    </item>
    ...
</blockList>
```

Note that the SDFRule includes a `<notPrev>` condition. This condition states that the number or letter or roman number must be tagged as an index unless the symbol "$*$" occurs in the previous word. In such a case, there is a footnote, not an index.

[11]`<isNumber>` is an optional feature, which will be explained in subsection 4.7 below. For now, it is sufficient to know that `<isNumber>true</isNumber>` evaluates to true iff the word only contains digits.

```
<SDFRule priority="10000" id="2">
    <tag>index</tag>
    <headAlternatives>
        <head>
            <isNumber>true</isNumber>
        </head>
        <head>
            <Bag name="Letters" />
        </head>
        <head>
            <Bag name="Roman Numbers" />
        </head>
    </headAlternatives>
    <nextAlternatives>
        <next maxDistance="1">
            <tag>index</tag>
            <headAlternatives>
                <head>
                    <Form>.</Form>
                </head>
                <head>
                    <Form>)</Form>
                </head>
            </headAlternatives>
        </next>
    </nextAlternatives>
    <prevStarAlternatives>
        <prev maxDistance="1">
            <tag>index</tag>
            <headAlternatives>
                <head>
                    <Form>(</Form>
                </head>
            </headAlternatives>
        </prev>
    </prevStarAlternatives>
    <prevAlternatives>
        <notPrev maxDistance="1">
            <headAlternatives>
                <head>
                    <Form>*</Form>
                </head>
            </headAlternatives>
        </notPrev>
    </prevAlternatives>
</SDFRule>
```

Of course, it is impossible to codify all potential exceptions within a single SDFRule, in terms of extra conditions like the `<notPrev>` condition of the SDFRule above. Indeed, even if it would be possible, the resulting SDFRule would be too big and so unreadable.

A better idea is to add extra SDFRule(s) *with higher priority, i.e., with priority greater than 10000*, tagging as "not-index" the words that are part of an exception. The `CJEUprocessorDEMO` assumes that all "not-index" tags *nullify* the "index" ones on the same words, so that an itemization will *not* be created for them.

Note that, in this way, it is also possible to handle exceptions of exceptions (and exceptions of exceptions of exceptions, etc.): it is sufficient to add SDFRule(s) with even higher priorities that assign "not-not-index" tags, which nullify the "not-index" ones.

An example of SDFRule that identifies an exception of "index" is the following:

```
<SDFRule priority="10001" id="5">
    <tag>not-index</tag>
    <headAlternatives>
        <head>
            <Form>s</Form>
        </head>
    </headAlternatives>
    <nextAlternatives>
        <next maxDistance="1">
            <headAlternatives>
                <head>
                    <Form>)</Form>
                </head>
            </headAlternatives>
        </next>
    </nextAlternatives>
    <prevAlternatives>
        <prev maxDistance="1">
            <headAlternatives>
                <head>
                    <Form>(</Form>
                    <blanksBefore>0</blanksBefore>
                </head>
            </headAlternatives>
        </prev>
    </prevAlternatives>
</SDFRule>
```

This SDFRule understands when the letter "s" is used to pluralize other words, e.g., in "place(s)". In such a case, of course it is not an index. The SDFRule checks whether "s" is surrounded by open and close round brackets and the open round bracket is attached without blanks to the previous word (`<blanksBefore>0</blanksBefore>`). Requiring zero blanks between the open round bracket and its previous word allows to distinguish, for instance, "place(s)" from "place (s)".

4.6 Well-formed SDFRule(s)

As explained above, SDFRule(s) are written in XML, but the XML format cannot be directly executed by the SDFTagger: SDFRule(s) need to be compiled in a special byte-code, which is then loaded and executed by the SDFTagger.

When compiling an SDFRule, the constructor of the class `convert2SDFCode.java` checks whether the XML format in input is well-formed. In case it is, the SDFCode is generated. In case it is not, `convert2SDFCode.java` throws an exception, bearing one of the messages reported below.

Error #1: `id`, `priority`, `maxDistance`, `maxHeight`, and `maxDepth` must be integers greater than zero!

Attribute(s) `id` and `priority` of `<SDFRule>` must be integers greater than zero, as well Attribute `maxDistance` of `<(not)Prev>` and `<(not)Next>`, Attribute `maxHeight` of `<(not)Governor>`, and Attribute `maxDepth` of `<(not)Dependent>`.

Error #2: AAA is a mandatory Attribute of EEE.

Each `<SDFRule>` Element must specify Attribute(s) `id` and `priority`; `<SDFRule>` does not allow any other Attribute. Each `(not)Prev` and each `(not)Next` must specify the Attribute `maxDistance`. Each `(not)Governor` must specify the Attribute `maxHeight`. Each `(not)Dependent` must specify the Attribute `maxDepth`. No other Attribute is allowed on these Element(s).

Error #3: Element EEE1 not allowed within Element EEE2.

The Element(s) within `<SDFRule>` allow a very limited set of sub-Elements:

- The Element(s) `<SDFRule>`, `<(not)Prev>`, `<(not)Next>`, `<(not)Governor>`, and `<(not)Dependent>` may only enclose sub-Element(s) `<tag>`, `<headAlternatives>`, `<prevStarAlternatives>`, `<prevAlternatives>`, `<nextStarAlternatives>`, `<nextAlternatives>`, `<governorAlternatives>`, and `<dependentsAlternatives>`.
- Element `<headAlternatives>` only allows `<head>`.
- Element `<prevAlternatives>` only allows `<(not)Prev>`.
- Element `<prevStarAlternatives>` only allows `<Prev>`.
- Element `<nextAlternatives>` only allows `<(not)Next>`.
- Element `<nextStarAlternatives>` only allows `<Next>`.
- Element `<governorAlternatives>` only allows `<(not)Governor>`.
- Element `<dependentsAlternatives>` only allows `<Dependents>`.
- Element `<Dependents>` only allows `<(not)Dependent>`.

Note that `<prevAlternatives>` accepts as children both `<Prev>` and `<notPrev>` while `<prevStarAlternatives>` only accepts `<Prev>` (and similarly for `<nextAlternatives>` and `<nextStarAlternatives>`: the former accepts as children both `<Next>` and `<notNext>` while the latter only `<Next>`).

The reason is that `<prevStarAlternatives>` and `<nextStarAlternatives>` can be iteratively applied for multiple times. At every iteration, their children are re-applied from the leftmost or rightmost node that satisfied the previous iteration. `<notPrev>` and `<notNext>` requires the non-existence of certain nodes. For instance, in case we have a `<notPrev>` with `maxDistance` equal to 3, it is required the none of the previous three nodes satisfies the conditions in the `<notPrev>`. If this requirement is met, it is undecidable to understand from which of the three nodes the iteration should be re-applied. For this reason, `<notPrev>` and `<notNext>` are not allowed within `<prevStarAlternatives>` and `<nextStarAlternatives>` respectively.

Error #4: Element EEE must contain one (and only one) `<headAlternatives>`

The Element(s) `<SDFRule>`, `<(not)Prev>`, `<(not)Next>`, `<(not)Governor>`, and `<(not)Dependent>` must contain a single `<headAlternatives>`. On the other hand, `<prevAlternatives>`, `<prevStarAlternatives>`, `<nextAlternatives>`, `<nextStarAlternatives>`, `<governorAlternatives>`, and `<dependentsAlternatives>` are optional.

Error #5: `<headAlternatives>` **must contain at least one** `<head>`**.**

It is not allowed to have empty `<headAlternatives>`(s). Note, however, that we can have empty `<head>`(s), i.e. `<headAlternatives><head></head></headAlternatives>`.

Specifying empty `<head>`(s) is useful whenever we need to state that any SFDNode is fine (provided there is at least one). For instance, the following SDFRule tags with as "non-root" every SDFNode which is not a root:

```
<SDFRule id="1" priority="1">
  <tag>non-root</tag>
  <headAlternatives><head></head></headAlternatives>
  <governorAlternatives>
    <Governor maxHeight="1">
      <headAlternatives><head></head></headAlternatives>
    </Governor>
  </governorAlternatives>
</SDFRule>
```

In other words, *any* SDFNode who has a governor (and *any* governor is fine) must be tagged with the tag "non-root".

Similarly, note that, by using a `notGovernor` Element, we can require the non-existence of SDFNode(s) along one of the four directions. For instance, the following SDFRule tag as "root" all SDFNode(s) that do *not* have a governor, i.e., for which there is not an SDFNode (and we do not accept *any* SDFNode) above.

```
<SDFRule id="1" priority="1">
  <tag>root</tag>
  <headAlternatives><head></head></headAlternatives>
  <governorAlternatives>
    <notGovernor maxHeight="1">
      <headAlternatives><head></head></headAlternatives>
    </notGovernor>
  </governorAlternatives>
</SDFRule>
```

On the other hand, the following SDFRule tag as "leaf" all SDFNode(s) that do *not* have dependents, i.e., for which there is not an SDFNode (and we do not accept *any* SDFNode) below.

```
<SDFRule id="1" priority="1">
  <tag>leaf</tag>
  <headAlternatives><head></head></headAlternatives>
  <dependentsAlternatives>
    <Dependents>
      <notDependent maxDepth="1">
        <headAlternatives><head></head></headAlternatives>
      </notDependent>
    </Dependents>
  </dependentsAlternatives>
</SDFRule>
```

Error #6: EEE1 must contain at least one EEE2.

Similarly to Error #5, `<prevStarAlternatives>` and `<prevAlternatives>` must contain at least one `<(not)Prev>`; `<nextStarAlternatives>` and `<nextAlternatives>` must contain at least one `<(not)Next>`; `<governorAlternatives>` must contain at least one `<(not)Governor>`; finally, `<dependentsAlternatives>` must contain at least one `<Dependents>`; and, `<Dependents>` must contain at least one `<(not)Dependent>`.

Error #7: Attribute XXX not allowed on Element YYY.

The Element `<SDFRule>` and its sub-Element(s) allow a very limited sets of attributes, each having very limited set of possible values:

- Element `<SDFRule>` only allows the Attribute(s) `id` and `priority`.
- Element(s) `<(not)Prev>` and `<(not)Next>` only allow the Attribute(s) `maxDistance`.
- Element `<(not)Governor>` only allows the Attribute(s) `maxHeight`.
- Element `<(not)Dependent>` only allows the Attribute(s) `maxDepth`.

Error #8: Value ZZZ not allowed on Attribute `endOfSentence`.

The Attribute `endOfSentence` only allows the (string!) values “true” and “false”.

Error #9: unforseen exception.

A standard Java exception, not included in the list above, is thrown.

4.7 SDFNodeConstraints

SDFNodeConstraints is a special class that allows to implement additional optional features of SDFHead, such as `<blanksBefore>` and `<isNumber>` seen above.

To this end, SDFNodeConstraints needs to be *extended* in a subclass that implements the checks on the optional features. The extended class is instantiated within the subclass of SDFTaggerConfig that will be given as parameter of SDFTagger’s constructor. If the SDFTaggerConfig given in input to the SDFTagger’s constructor instantiates the SDFNodeConstraints basic class, rather than an extended class, the SDFTagger will not consider any optional feature (i.e., only mandatory features are checked).

For instance, as said above in subsection 4.1, the `CJEUprocessorDEMO` instantiates three SDFTagger(s), each of which on a different subclass of SDFTaggerConfig. These are the classes SDFTaggerConfigForStructuralTagging, SDFTaggerConfigForPartyClassificationOnKeywords, and SDFTaggerConfigForEntityLinking; the first two instantiate two subclasses of SDFNodeConstraints, i.e., SDFNodeConstraintsForStructuralTagging and SDFNodeConstraintsForClassificationOnKeywords, while the third one instantiates the basic class SDFNodeConstraints, in that it does not use optional features.

The instances of SDFNodeConstraints or its subclasses are assigned to the SDFTaggerConfig’s attribute “SDFNodeConstraintsFactory”; for instance, the class SDFNodeConstraintsForStructuralTagging contains the following statement:

SDFNodeConstraintsFactory = new SDFNodeConstraintsForStructuralTagging();

SDFNodeConstraintsFactory takes its name from the fact that a new instance of its value will be created and used in the SDFRule(s) executed by the SDFTagger, for each `headAlternatives` of the SDFRule.

Among the protected methods of SDFNodeConstraints only two of them need to be overridden in the subclasses. Some of the other methods are protected in that they need to be called from classes belonging to the same package. The first method is:

protected SDFNodeConstraints FactorySDFNodeConstraints(){*...*}

which needs to be overridden in order to create and return an instance of the subclass of SDFNodeConstraints. For instance, the class SDFNodeConstraintsForStructuralTagging overrides this method as follows:

protected SDFNodeConstraints FactorySDFNodeConstraints()

{*return new SDFNodeConstraintsForStructuralTagging();*}

The second protected method that needs to be overridden is:

```
protected boolean checkOptionalFeatures(Hashtable<String, String> optionalFeatures,
                                        SDFHead SDFHead){...}
```

The user must override this method in order to scan the list of optional features and, for each of them, implement suitable checks for determining whether the feature is true or false in the given SDFHead.

Since the optional features are given in input as an Hashtable<String, String>, a practical way to check them is to implement a 'while' cycle in the form:

```
Enumeration en = optionalFeatures.keys();
while(en.hasMoreElements())
{
        String key = (String)en.nextElement();
        if(key.compareToIgnoreCase("blanksBefore")==0){...}
        else if(key.compareToIgnoreCase("isNumber")==0){...}
                ...
}
```

Every inner 'if' condition of the while cycle implements the check of an optional feature.

In most cases, the method simply returns false iff the two values are different (or the SDFHead does not include it). An example is `<blanksBefore>`, which is a feature assigned during the parsing of the original input text (see subsection 3.2 above):

```
if(key.compareToIgnoreCase("blanksBefore")==0)
{
    String blanksBeforeSDFHead = SDFHead.getOptionalFeaturesValue(key);
    if(blanksBeforeSDFHead==null)return false;
    if(optionalFeatures.get(key).compareToIgnoreCase(blanksBeforeSDFHead)!=0)
                                                                return false;
}
```

However, more complex checks are possible; for instance, the optional feature `<isNumber>` requires to check iff each character of the Form is a digit, in case the feature has value `<true>`, or that at least one character of the form is not a digit, in case the feature has value `<false>`:

```
if(key.compareToIgnoreCase(”isNumber”)==0)
{
    String isNumber = optionalFeatures.get(key);
    if(isNumber.compareToIgnoreCase(”true”)==0)
    {
        for(int i=0;i<SDFHead.getForm().length();i++)
            if(Character.isDigit(SDFHead.getForm().charAt(i))==false)
                return false;
    }
    else if(isNumber.compareToIgnoreCase(”false”)==0)
    {
        for(int i=0;i<SDFHead.getForm().length();i++)
            if(Character.isDigit(SDFHead.getForm().charAt(i))==true)
                return false;
    }
    else return false;
}
```

Other examples may be directly seen in the `CJEUprocessorDEMO` Java code.

4.8 SDFLogger

SDFLogger monitors the SDFRule(s) enforced by an instance of SDFTagger, and stores the step-by-step traces of these SDFRule(s) within a "readable" XML file.

SDFLogger is instantiated within SDFTaggerConfig. The constructor of SDFTaggerConfig takes as parameter the Java File where SDFLogger will write the step-by-step traces, which is in turn passed to the constructor of SDFLogger, when this is instantiated. In case the SDFTaggerConfig's parameter is "null", no logging of the executions will be carried out by SDFLogger.

The `CJEUprocessorDEMO` has two files having the 'main' Java method. These are `buildAkomaNtosoCJEUs.java` and `buildAkomaNtosoCJEUsDEBUG.java`. The former processes the whole corpus of case law of the European Court of Justice (CJEU), and does not have a log file (all instances of SDFTaggerConfig are created by passing "null" as parameter). On the other hand, `buildAkomaNtosoCJEUsDEBUG.java` processes a single file of the corpus, specified in the Java code, and defines two[12] log files, which are passed to the corresponding instances of SDFTaggerConfig and so to the SDFLogger(s) instantiated therein:

File logFileForStructuralTagging =
new File("./CORPUS/LogFiles/SDFDebugForStructuralTagging.xml");

File logFileForPartyClassificationOnKeywords =
new File("./CORPUS/LogFiles/SDFDebugForPartyClassificationOnKeywords.xml");

The log file is structured as follows:

```
<SDFDebugs>
    <SDFDebug>
        <SDFNodes>...</SDFNodes>
        <SDFRules>...</SDFRules>
        <SDFTraces>...</SDFTraces>
        <SDFTags>...</SDFTags>
    </SDFDebug>
        ...
    <SDFDebug>
        <SDFNodes>...</SDFNodes>
        <SDFRules>...</SDFRules>
        <SDFTraces>...</SDFTraces>
        <SDFTags>...</SDFTags>
    </SDFDebug>
</SDFDebugs>
```

[12]No log file is created for SDFTaggerConfigForEntityLinking. As it will be explained in section 6 below, the SDFTagger for entity linking first creates automatically the SDFRule(s) for recognizing the parties in text, and then executes them. This is done file-by-file, so that an *array* of such SDFTagger(s) is indeed created. Therefore, the management of the log file(s) for entity linking is much more complex. The `CJEUprocessorDEMO` avoids it, in that it only aims at explaining how the SDFTagger system works.

Each `<SDFDebug>` shows the step-by-step traces of a set of SDFRule(s) (reported in `<SDFRules>`) on a specific chain[13] of SDFNode(s) (reported in `<SDFNodes>`). Some of these SDFRule(s) succeed, and assign the SDFTag(s) (reported in `<SDFTags>`) to the SDFNode(s). The step-by-step traces of the SDFRule(s), both the ones that succeeded and the ones that failed, is shown in `<SDFTraces>`.

As an example, consider the following trace, taken from the file SDFDebugForStructuralTagging.xml:

```
<Step onSDFNode="1" id="23" idInstance="17" priority="10000" result="OK">
   1     <prevAlternatives>
   2         <Step onSDFNode="none" result="OK" />
   3     </prevAlternatives>
   4     <nextStarAlternatives>
   5         <Step onSDFNode="2" result="FAILED" />
   6         <Step onSDFNode="2" result="OK" />
   7         <Step onSDFNode="3" result="FAILED" />
   8         <Step onSDFNode="3" result="OK" />
   9         <Step onSDFNode="4" result="FAILED" />
   10        <Step onSDFNode="4" result="OK" />
   11        <Step onSDFNode="5" result="FAILED" />
   12        <Step onSDFNode="5" result="OK" />
   13        <Step onSDFNode="6" result="FAILED" />
   14        <Step onSDFNode="6" result="OK" />
   15        <Step onSDFNode="7" result="FAILED" />
   16        <Step onSDFNode="7" result="OK" />
   17        <Step onSDFNode="8" result="OK">
   18            <nextStarAlternatives>
   19                <Step onSDFNode="9" result="OK" />
   20                <Step onSDFNode="10" result="OK" />
   21                <Step onSDFNode="11" result="OK" />
   22                <Step onSDFNode="12" result="OK" />
   23                <Step onSDFNode="13" result="OK" />
   24                <Step onSDFNode="14" result="OK" />
   25                <Step onSDFNode="15" result="OK" />
   26                <Step onSDFNode="16" result="FAILED" />
   27            </nextStarAlternatives>
   28        </Step>
   29        <Step onSDFNode="16" result="FAILED" />
   30        <Step onSDFNode="16" result="FAILED" />
   31    </nextStarAlternatives>
</Step>
```

The trace refers to the step-by-step execution of the following SDFRule, devoted to identify titles in the CJEU case law:

[13] In the `CJEUprocessorDEMO`, the chain of SDFNode(s) is always the whole chain of SDFNode(s)/words in input, in that the input files include a single `<DependencyTrees>`.

```
<SDFRule priority="10000" id="23">
  <tag>tblock</tag>
  <headAlternatives>
    <head><Font>title</Font></head>
  </headAlternatives>
  <nextStarAlternatives>
    <next maxDistance="1">
      <tag>tblock</tag>
      <headAlternatives>
        <head>
          <Font>title</Font>
          <endOfSentence>true</endOfSentence>
        </head>
      </headAlternatives>
      <nextStarAlternatives>
        <next maxDistance="1">
          <headAlternatives>
            <head><Font>title</Font></head>
          </headAlternatives>
        </next>
      </nextStarAlternatives>
    </next>
    <next maxDistance="1">
      <tag>tblock</tag>
      <headAlternatives>
        <head><Font>title</Font></head>
      </headAlternatives>
    </next>
  </nextStarAlternatives>
  <prevAlternatives>
    <notPrev maxDistance="1">
      <headAlternatives>
        <head />
      </headAlternatives>
    </notPrev>
    <prev maxDistance="1">
      <headAlternatives>
        <head><Font>normal</Font></head>
      </headAlternatives>
    </prev>
    <prev maxDistance="1">
      <headAlternatives>
        <head><endOfSentence>true</endOfSentence></head>
      </headAlternatives>
    </prev>
  </prevAlternatives>
</SDFRule>
```

The SDFRule is successful (`result="OK"`) on the SDFNode with `<index>1</index>`, i.e., the SDFNode with the same index specified in the `<Step>`'s attribute `onSDFNode`; note that all `<SDFNodes>`(s) in `<SDFNodes>` are indexed.

SDFTagger executes the SDFRule along the steps indicated in the trace. First, it executes its `<prevAlternatives>` condition; this is successful on its first subcondition:

```
<notPrev maxDistance="1">
    <headAlternatives>
        <head />
    </headAlternatives>
</notPrev>
```

This condition requires the non-existence of a previous SDFNode. It is successful in that the SDFNode is the first one, so that no previous SDFNode exists:

```
2    <Step onSDFNode="none" result="OK" />
```

Then, SDFTagger executes the `<nextStarAlternatives>` condition. On the SDFNode that immediately follows the one with `<index>1</index>`, the first `<next>` of the `<nextStarAlternatives>` fails, in that the SDFNode does not end a sentence; however, the second `<next>` of the `<nextStarAlternatives>` condition succeeds. Therefore, the `<nextStarAlternatives>` is executed again... and again and again, as it continues to succeed on its second `<next>` until the SDFNode with `<index>7</index>`:

```
5        <Step onSDFNode="2" result="FAILED" />
6        <Step onSDFNode="2" result="OK" />
7        <Step onSDFNode="3" result="FAILED" />
8        <Step onSDFNode="3" result="OK" />
9        <Step onSDFNode="4" result="FAILED" />
10       <Step onSDFNode="4" result="OK" />
11       <Step onSDFNode="5" result="FAILED" />
12       <Step onSDFNode="5" result="OK" />
13       <Step onSDFNode="6" result="FAILED" />
14       <Step onSDFNode="6" result="OK" />
15       <Step onSDFNode="7" result="FAILED" />
16       <Step onSDFNode="7" result="OK" />
```

Then, the first `<next>` of the `<nextStarAlternatives>` condition succeeds on the SDFNode with `<index>8</index>`, in that the SDFNode indeed ends the sentence. Note that the first `<next>` includes another (inner) `<nextStarAlternatives>`, which is satisfied until words marked as "title" are found, i.e., until the SDFNode with `<index>15</index>`:

```
    <Step onSDFNode="8" result="OK">
        <nextStarAlternatives>
            <Step onSDFNode="9" result="OK" />
            <Step onSDFNode="10" result="OK" />
            <Step onSDFNode="11" result="OK" />
            <Step onSDFNode="12" result="OK" />
            <Step onSDFNode="13" result="OK" />
            <Step onSDFNode="14" result="OK" />
            <Step onSDFNode="15" result="OK" />
            <Step onSDFNode="16" result="FAILED" />
        </nextStarAlternatives>
    </Step>
```

Note that the inner `<nextStarAlternatives>` *does not assign any tag.* Its role is simply the one of "moving the next-cursor" until the last word marked as "title", in order to avoid the second `<next>` of the (outer) `<nextStarAlternatives>` to tag them as "tblock". This is indeed the desired result, as they belong to another title, and so they have to be marked as "tblock" by a *different* SDFRule.

Finally, both `<next>`(s) of the (outer) `<nextStarAlternatives>` fail on the SDFNode with `<index>16</index>`:

```
    <Step onSDFNode="16" result="FAILED" />
    <Step onSDFNode="16" result="FAILED" />
```

The SDFRule concludes its execution and tags the SDNode(s) from `<index>1</index>` to `<index>8</index>` with the tag "tblock". This is reported in the `<SDFTags>` section of `</SDFDebug>`:

```
<SDFTag tag="tblock" id="23" idInstance="17" ... onSDFNode="1" />
<SDFTag tag="tblock" id="23" idInstance="17" ... onSDFNode="2" />
<SDFTag tag="tblock" id="23" idInstance="17" ... onSDFNode="3" />
<SDFTag tag="tblock" id="23" idInstance="17" ... onSDFNode="4" />
<SDFTag tag="tblock" id="23" idInstance="17" ... onSDFNode="5" />
<SDFTag tag="tblock" id="23" idInstance="17" ... onSDFNode="6" />
<SDFTag tag="tblock" id="23" idInstance="17" ... onSDFNode="7" />
<SDFTag tag="tblock" id="23" idInstance="17" ... onSDFNode="8" />
```

5 KBInterface

The KBInterface package includes an homonym Java interface, which is implemented by the classes XMLFilesManager and MongoDBManager.

XMLFilesManager loads and manages the SDFRule(s) from files, while MongoDBManager (which indeed extends XMLFilesManager) loads the SDFRule(s) from files and populates a database in the MongoDB[14] DBMS, so that the management of the SDFRule(s) is later done through MongoDB. Loading and transferring the data from/to MongoDB avoids keeping the SDFRule(s) in the local memory, a choice that could be necessary when SDFTagger has to manage a large set of SDFRule(s).

Further Java classes that implement the interface KBInterface may be built in the future as well, in order to manage SDFRule(s) from/to alternative DBMSs or storage mechanisms. The SDFTagger library, however, at the moment offers only Java classes that implement KBInterface from/to files or MongoDB.

As previously explained, SDFRule(s) are written, either manually or automatically, within XML files that are compiled in a special bytecode, before being executed. Whenever a new instance of SDFTagger is created, XMLFilesManager automatically checks whether some XML files have been updated and, in such a case, it recompiles them.

SDFTaggerConfig includes an attribute KBManager, of type KBInterface, to be instantiated in its subclasses. For instance, the three subclasses of SDFTaggerConfig in the `CJEUprocessorDEMO`, which (all) use XMLFilesManager, instantiate it as follows:

KBManager = new XMLFilesManager(
rootDirectoryBags, localPathsBags, rootDirectoryXmlSDFRules,
rootDirectoryCompiledSDFRules, localPathsSDFRules);

The five parameters of the XMLFilesManager's constructor are also attributes of SDFTaggerConfig that need to be specified in the subclasses:

- **rootDirectoryBags** and **localPathsBags** specify the root directory of the XML files that contain the bags and the (relative) paths of these files.

- **rootDirectoryXmlSDFRules** and **localPathsSDFRules** specify the root directory of the XML files that contain the SDFRule(s) and the (relative) paths of these files.

- **rootDirectoryCompiledSDFRules** specifies the root directory where XMLFilesManager will write the compiled files, while following the same directory structure of their XML counterparts.

[14]`https://www.mongodb.com`

When SDFTagger is instantiated, XMLFilesManager checks whether the XML representations of the SDFRule(s) have been updated with the respect to their compiled versions. In case of misalignment between the two, it compiles the XML representations and generates the (new) compiled files, by possibly rewriting the old ones. On the other hand, MongoDBManager does not automatically check whether the SDFRule(s) have been updated: in such a case, the user must check them "manually" and possibly reload the MongoDB in case they have been updated.

Once SDFRule(s) are properly compiled and loaded, KBManager scans the chain of SDFNode(s) and, for each SDFNode in the chain, it determines the bags to be inserted on the SDFNode as well as the set of SDFRule(s) that are triggered by that SDFNode. Finally, it passes that set of SDFRule(s) to the SDFTagger, which executes them.

6 The `CJEUprocessorDEMO`

The present user manual is provided together with a demo, the `CJEUprocessorDEMO`, that crawls, parses, and processes via SDFTagger, 625 case law from the European Court of Justice (CJEU). These case law are publicly available online through the CELLAR system administered by the Publications Office of the European Union[15].

The `CJEUprocessorDEMO` is provided as a free software under the terms of the GNU General Public License (version 3)[16]; the demo uses the SDFTagger library that may be likewise made available upon request (please contact me, if you are interested).

The `CJEUprocessorDEMO` consists in a pipeline of three components: A_CJEU_Crawler, B_CJEU_Parsing, and C_CJEU_AkomaNtoso. The first two components respectively crawl the HTML files of the case law from the Web and parse them with the Stanford CoreNLP[17], version 3.8.0. The present user manual skips the description of these two components; the comments in the Java files should suffice for the user to download and parse[18] the HTML files again.

The files are stored within the "./CORPUS" subfolder; after executing the component B_CJEU_Parsing, the subfolder "CORPUS/1 - PARSED INPUT" includes the parsed files, in the XML format shown and discussed above in section 3.

The C_CJEU_AkomaNtoso component processes via SDFTagger the files in "CORPUS/1 - PARSED INPUT" and builds corresponding XML files in the Akoma Ntoso legal standard[19], which are then stored in the "CORPUS/2 - AKOMA NTOSO" subfolder.

There are two Java files in the C_CJEU_AkomaNtoso component having the 'main' method: `buildAkomaNtosoCJEUs.java` and `buildAkomaNtosoCJEUsDEBUG.java`. The former processes all files in "CORPUS/1 - PARSED INPUT" while the latter only the one specified in its main attribute "fileName".

The two files have another boolean attribute "resetAll". If it is set to true, all files in "CORPUS/2 - AKOMA NTOSO" are deleted before generating new ones; on the other hand, if it is set to false, the generated files are only added to the folder without deleting the ones already there (but in case of overriding).

Both `buildAkomaNtosoCJEUs.java` and `buildAkomaNtosoCJEUsDEBUG.java` create and use three instances of SDFTagger, each on a different configuration file:

- **SDFTaggerConfigForStructuralTagging**, needed to identify the structure of the case law, i.e., sections, subsections, paragraphs, titles, itemizations, etc.;
- **SDFTaggerConfigForPartyClassificationOnKeywords**, needed to classify the parties of a case law as either company or institution, once the words in the party's proper name have been identified;
- **SDFTaggerConfigForEntityLinking**, needed to identify the parties classified by SDFTaggerConfigForPartyClassificationOnKeywords in the rest of the text, either through their standard name or through a variant of their standard name.

[15]`https://publications.europa.eu/en/home`

[16]`https://www.gnu.org/licenses/gpl-3.0.html`

[17]`https://stanfordnlp.github.io/CoreNLP`

[18]IMPORTANT! The Stanford CoreNLP takes a lot of time to parse the HTML files! Be aware of that, in case you want to run the `convertXMLfiles.java` file ;-)

[19]`http://www.akomantoso.org`

It is worth noticing that SDFTaggerConfigForEntityLinking deals with SDFRule(s) that are *automatically* generated from the annotations made by SDFTaggerConfigForPartyClassificationOnKeywords. Moreover, the Akoma Ntoso output files are enriched by connecting each occurrence of the companies and institutions identified in the text with individuals in a simple reference OWL ontology, which is stored in the subfolder "CORPUS/OWL_Ontology". The ontology individuals are also created during processing.

The subsections below quickly describe each of the three SDFTagger instantiations; further comments to enhance comprehension are directly available in the Java code.

IMPORTANT!!! The aim of the `CJEUprocessorDEMO` is only the one of explaining how to use the SDFTagger system. It is not intended to be exhaustive with respect to the processing of the CJEU case law. I consider the result as "satisfactory" but far from being "accurate": further SDFRule(s) should be added to this end. I have no intention to carefully inspect the output Akoma Ntoso files and add these SDFRule(s), as a kind of "hobby" or because you think I've the duty to do it. So please don't bother me by notifying me wrong or missing annotations. You can rather fix them yourself, it would be a very good exercise to better learn how to use the SDFTagger system ;-)

6.1 SDFTaggerConfigForStructuralTagging

The SDFTagger instantiated on this configuration identifies the structural elements provided by the Akoma Ntoso standard for representing case law. This SDFTagger takes in input the syntactic representations produced by the Stanford CoreNLP and stored within the XML files in "CORPUS/1 - PARSED INPUT".

Each of these files includes a single big `<DependencyTrees>` XML element, whose children are the `<DependencyTree>`(s) identified by the parser on the input text. The input text is considered as a whole; in other words, the HTML structure of the case law is basically ignored: the `CJEUprocessorDEMO` considers the input text as it was a single (long) line of text, obtained, for instance, by cut&pasting the text into a ".txt" file. Note, however, that some information from the HTML file is kept, and codified in the `<DependencyTree>`(s) within the feature `<Font>`. Specifically, the words marked in the HTML as a title, a signature, or in bold (bold is used to identify titles and parties, see below) are codified as specific values of the `<Font>` feature, for being later matched by corresponding optional features of the SDFRule(s).

For each case law, this SDFTagger identifies the three main Akoma Ntoso sections of the case law (`<header>`, `<judgmentBody>`, `<conclusions>`), as well as the four subsections of `<judgmentBody>` (`<introduction>`, `<background>`, `<motivation>`, `<decision>`). Furthermore, within each subsection, it identifies the numbered paragraphs in which the CJEU case law are structured (encoded in the Akoma Ntoso element `<paragraph>`), itemizations and enumerations (encoded in the Akoma Ntoso element `<blockList>`), and the titles (encoded in the Akoma Ntoso element `<tblock>`).

Finally, this SDFTagger identifies the *parties* involved in the trial, each of which will be encoded in the Akoma Ntoso element `<party>`. Parties are marked in bold in the input HTML file, so that the SDFRule(s) of this SDFTagger exploit the optional feature `<Font>` to identify them. There are three categories of parties: appellant, respondent, and intervening. SDFRule(s) are able to distinguish the identified parties in the three categories by looking at special keywords in the text surrounding them. The category is specified in the attribute "`as`" of the Akoma Ntoso element `<party>`.

The SDFRule(s) loaded and used by this SDFTagger are stored within the subfolder SDFTaggerConfigForStructuralTagging/SDFRulesXML of the C_CJEU_AkomaNtoso component. The SDFRule(s) identifying sections and subsections are stored within homonym XML files, except paragraphs which are identified with the SDFRule(s) in the file "indexes.xml". The SDFRule(s) in this file also identify itemizations and enumerations. Titles and parties are respectively identified by the SDFRule(s) in the files "titles.xml" and "parties.xml".

Once all the information above has been identified, an initial Akoma Ntoso representation of the case law is built and encoded in a JDOM[20] object.

6.2 SDFTaggerConfigForPartyClassificationOnKeywords

The SDFTagger instantiated on SDFTaggerConfigForPartyClassificationOnKeywords inspects the JDOM object from the previous SDFTagger and tries to classify all entities tagged as `<party>` as either company or institution. This is trivally done by searching some keywords among the words enclosed in the `<party>` XML tag.

These keywords are directly codified in the SDFRule(s) belonging to "companies.xml" and "institutions.xml", which are located in the SDFTaggerConfigForPartyClassificationOnKeywords/SDFRulesXML subfolder of the C_CJEU_AkomaNtoso component.

For the parties that are recognized as either company or institution, a new individual is created in the ontology. Individuals will be connected with all Akoma Ntoso annotations of their mentions: both the mention within the `<party>` tag itself and the other mentions in the rest of the text, which will be tagged with the Akoma Ntoso element `<organization>`. Both `<party>` and `<organization>` use the attribute "`refersTo`" to connect the ontology individuals through the `<meta>` section of the Akoma Ntoso file.

Mentions of companies and institutions are recognized via the SDFRule(s) enforced by the SDFTagger instantiated on SDFTaggerConfigForEntityLinking (see next subsection).

On the other hand, in case this SDFTagger does not manage to classify the parties as either company or institution, the "`refersTo`" attribute of `<party>` is left empty.

6.3 SDFTaggerConfigForEntityLinking

The SDFTagger instantiated on SDFTaggerConfigForEntityLinking retrieves from the outcome of the previous SDFTagger all parties that have been classified as either company or institution and generates all SDRule(s) able to recognize them in text. Then, it executes these SDRule(s) in all textual excerpts belonging to both the present case law

[20]`http://www.jdom.org`

and the ones belonging to all subsequent ones, in order to link the same entities to the same ontological individuals, in view of enabling cross-document navigation and search.

For each company or institution, one or more SDFRule(s) are (automatically!) generated: a basic SDFRule recognizing the name of the party exactly as it is, plus additional SDFRule(s) able to recognize *variants* of this name.

At present, the `CJEUprocessorDEMO` "expands" the basic SDFRule only for institutions, while companies are recognized in text only if they appear exactly as they are mentioned in the Akoma Ntoso XML tag `<party>`.

SDFRule(s) are created by the classes in the "OntologyManager/SDFBuilder" package of `CJEUprocessorDEMO`, which are called after individuals are created in the ontology.

The remaining of the section focuses on the `InstitutionsSDFRulesBuilder.java` class in particular, while exemplifying its implementation and outcome on the specific institution "Autorità nazionale anticorruzione (ANAC)"[21], which is recognized as party in the CJEU case law "ECLI_EU_C_2017_1000.xml". In order to better understand which SDFRule(s) are created for this institution, I advise the user to run the mentioned file alone, via `buildAkomaNtosoCJEUsDEBUG.java`.

Assuming that the individual "institution_1" has been created in the ontology for this institution, the method "createSDFRulesForInstitutionFromSDFHeads" of the class `InstitutionsSDFRulesBuilder.java` is called, in order to create the SDFRule(s) recognizing the institution in text.

The first SDFRule that is created, i.e., the one that has been called above as "the basic SDFRule", simply recognizes the input text exactly as it is. This is obtained by taking the first word ("autorità") as trigger word of the SDFRule and by concatenating all the other words after it. The `head`(s) of the SDFRule only checks the `Form` of the words and the `next`(s) are all asserted with `maxDistance`=1. Figure 6 shows a graphical representation of the basic SDFRule for the institution under examination.

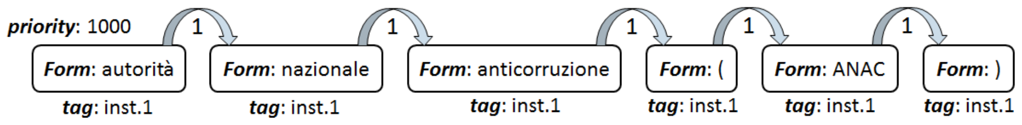


Figure 6: The basic SDFRule for "Autorità nazionale anticorruzione (ANAC)"

The SDFRule in Figure 6 tags all the words in the sequence with the name of the individual ("institution_1"), which is of course unique in the ontology, in order to create the ontological link, whenever the SDFRule succeeds.

Of course this SDFRule is very strict, first of all because it checks that the string is ended by the sequence "("+"ANAC"+")", i.e., the acronym of the institution enclosed between round brackets.

This is rather common in CJEU case law: whenever these case law mention institutions among the parties, they report their acronyms together with the full names. Then, in the rest of the case law, the institution is usually mentioned via its acronym.

[21]`http://www.anticorruzione.it`

In light of this, the `InstitutionsSDFRulesBuilder.java` class includes a method "searchForAcronym" that searches whether the basic SDFRule ends with a pattern in the form "("+"XXX"+")", where "XXX" is of course an acronym. If that pattern is found, the method creates a copy of the basic SDFRule, separates the acronym from the full name, and *adds other two SDFRule(s)*: one recognizing the full name only and one recognizing only the acronym. In other words, at the end of the method, *three* SDFRule(s) are generated for this institution: the basic one and the two ones depicted in Figure 7.

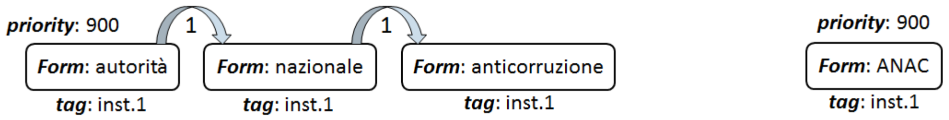


Figure 7: Two additional SDFRule(s) for "institution_1"

Note that the priority of the two additional SDFRule(s) in Figure 7 is *lower* than the one of the basic SDFRule. SDFTagger must indeed give precedence to the SDFRule in Figure 6 over the two ones in Figure 7. In fact, when the full name of the institution appears in text together with its acronym among parentheses, SDFTagger must create a single annotation. In other words, SDFTagger must create this annotation:

`<organization refersTo="institution_1">`Autorità nazionale anticorruzione (ANAC)`</organization>`

rather than the following two:

`<organization refersTo="institution_1">`Autorità nazionale anticorruzione`</organization>`
(`<organization refersTo="institution_1">`ANAC`</organization>`)

Actually, the criterion about the priorities of the SDFRule(s) just exemplified is a general one: the more an SDFRule is "strict", the higher should be its priority.

In light of this, we could "relax" the SDFRule(s) into additional lower-priority SDFRule(s), in order to encompass for typos or similar spellings. For instance, indeed "institution_1" could be also mentioned in some texts as "Autorità nazionale *per l'* anticorruzione" (translated in English as "national authority *for the* anticorruption"), obtained by adding the preposition "per" ("for") and the article "l'" ("the") between the words "nazionale" and "anticorruzione".

The class `InstitutionsSDFRulesBuilder.java` automatically generates then a fourth SDFRule, shown in Figure 8, from the one recognizing the full name of the institution (Figure 7, on the left) by increasing the `maxDistance` to 3 and by considering the lemmas of the words, rather than their Form. This SDFRule is of course less strict than the ones above, so that the priority has to be decreased.

The SDFRule in Figure 8 may be then seen as a kind of "backup" SDFRule for the ones in Figure 7: in case these ones fail, as it would happen, for instance, in "Autorità

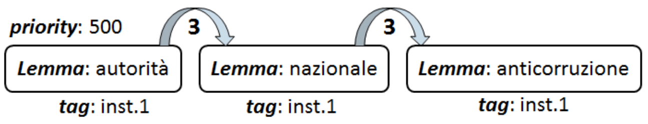


Figure 8: Another additional SDFRule for “institution_1”

nazionale *per l'* anticorruzione”, the one in Figure 8 would be still able to recognize the institution, although with a lower “weight”, i.e., with a lower priority.

In the same spirit, it is possible to further analyze the words in the SDFRule(s) created so far, and possibly crossing them with external sources such as thesauri, in order to generate a wider set of SDFRule(s) recognizing names of institutions in all the possible variants that may be thought. For instance, it may be observed that “anticorruzione” could be also written as “anti-corruzione”, i.e., with an hyphen “-” separating the prefix “anti” and the noun “corruzione”. To this end, the class `InstitutionsSDFRulesBuilder.java` includes a method “buildSDFRulesOnPrefixesAndSuffixes” that automatically searches for prefixes and suffixes of the words in the SDFRule(s) and generates the corresponding variants with the hyphen. In case of “institution_1”, it generates and adds to the knowledge base the two SDFRule(s) in Figure 9:

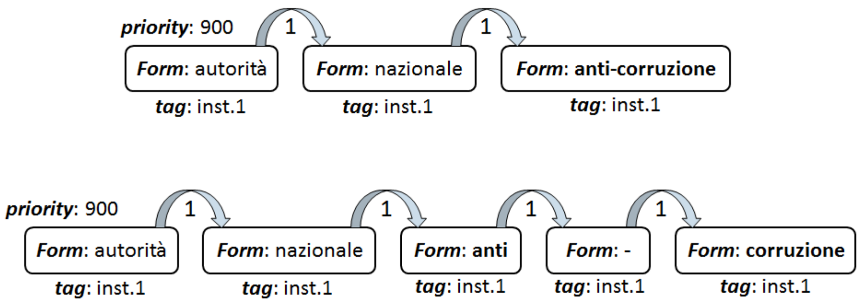


Figure 9: Other two additional SDFRule(s) for “institution_1”

Two SDFRule(s) are generated in Figure 9, rather than a single one, for being tolerant both with parsers that consider the three tokens as distinct words, and with parsers that consider them as a single one.

Finally, `InstitutionsSDFRulesBuilder.java` also generates SDFRule(s) that recognize the English name of non-English institutions, such as the one under examination, by exploiting Wikipedia. Specifically, the method “createSDFRuleOnEnglishNameFromWikipedia” in `InstitutionsSDFRulesBuilder.java` searches for the institution in Wikipedia (note that it is quite likely that institutions have an entry in Wikpedia), download the Wikipedia HTML page and searches therein for the link to the Wikipedia page in English. It then connects to the latter, where it finds the English name of the institution, from which an additional SDFRule is generated (Figure 10).

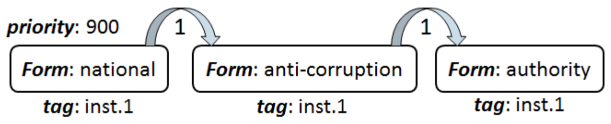


Figure 10: Another additional SDFRule for "institution_1"

Note that the English name of "autorità nazionale anticorruzione" features an hyphen after the prefix "anti", contrary to its Italian counterpart.

All SDFRule(s) shown from Figure 6 to Figure 10 are finally executed on all textual excerpts in the file "ECLI_EU_C_2017_1000.xml", so that all mentions of the same institution are annotated all over the case law. As said above, the SDFRule(s) are also executed on the subsequent files; these may also contain references to the same institution, which are then linked to same ontological individual.

Of course, other ad-hoc methods may be implemented in order to generate SDFRule(s) that recognize variants of companies name (file `CompaniesSDFRulesBuilder.java`). It is easy to understand that the generation of these SDFRule(s) is different from the generation of SDFRule(s) recognizing variants of institutions' names. For instance, many companies do not appear in Wikipedia; however, they usually have a website, which may be automatically linked to the ontological individuals.

On the other hand, in case we need to investigate relations between companies and their clients, as it is common, for instance, in know-your-customer (KYC) applications, the names of the companies' clients may be automatically downloaded from the website and searched in new texts. In addition, other sources at disposal of the user, e.g., registers from the chamber of commerce, news from the Web, etc. may be inspected and crossed with the information collected from the Web in order to automatically generate a wide set of SDFRule(s) able to identify relevant information, possibly by "weighting" them on the priorities of the SDFRule(s). The present manual should have made clear that the XML format of the SDFRule(s) is flexible enough to enable easy automatic generation of linguistic patterns to recognize in textual input; however, the generation of the SDFRule(s) depends on the specific application SDFTagger is used for.