

CFE User Guide

Authors: The Apache UIMA Development Community

Version 2.3.0

Incubation Notice and Disclaimer. Apache UIMA is an effort undergoing incubation at the Apache Software Foundation (ASF). Incubation is required of all newly accepted projects until a further review indicates that the infrastructure, communications, and decision making process have stabilized in a manner consistent with other successful ASF projects. While incubation status is not necessarily a reflection of the completeness or stability of the code, it does indicate that the project has yet to be fully endorsed by the ASF.

License and Disclaimer. The ASF licenses this documentation to you under the Apache License, Version 2.0 (the "License"); you may not use this documentation except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, this documentation and its contents are distributed under the License on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Trademarks. All terms mentioned in the text that are known to be trademarks or service marks have been appropriately capitalized. Use of such terms in this book should not be regarded as affecting the validity of the the trademark or service mark.

Table of Contents

1. Overview	1
1.1. Motivation	1
1.2. Approaches to feature extraction	2
1.2.1. Custom CAS Consumers	2
1.2.2. CFE approach	3
1.3. CFE Basics	3
2. Components	5
2.1. FESL XSD	5
2.2. Source Code	5
2.3. Descriptors	5
3. Configuration Files	7
3.1. Common notations and tags	7
3.1.1. Feature path	7
3.1.2. Full path and partial path	7
3.1.3. TAM and FAM	8
3.1.4. Arrays	8
3.1.5. Parent tag	9
3.1.6. Null values	9
3.1.7. Implicit TA exclusion	9
3.2. FESL Elements	10
3.2.1. BitsetFeaturaValuesXML	10
3.2.2. EnumFeatureValuesXML	10
3.2.3. ObjectPathFeatureValuesXML	11
3.2.4. PatternFeatureValuesXML	12
3.2.5. RangeFeatureValuesXML	12
3.2.6. SingleFeatureMatcherXML	13
3.2.7. GroupFeatureMatcherXML	14
3.2.8. PartialObjectMatcherXML	16
3.2.9. FeatureObjectMatcherXML	17
3.2.10. TargetAntotationXML	19
3.3. Configuration file sample	20
3.3.1. Task definition	20
3.3.2. Implementation	21
4. Using CFE for evaluation	27

Chapter 1. Overview

1.1. Motivation

Feature extraction, the extraction of information from data sources, is a common task frequently required to be performed by many different types of applications, such as machine learning, performance evaluation, and statistical analysis. This guide describes a tool that can be used to facilitate this extraction process, in conjunction with the Unstructured Information Management Architecture (UIMA), particularly focusing on text processing applications. UIMA provides a mechanism for executing modules called Analysis Engines that analyze artifacts (text documents in our case) and store the results of the analysis in a data structure called the Common Analysis Structure (CAS). These results are stored as `FeatTre Structures`, which are simply data structures that have an associated type and a set of properties in the form of attribute/value pairs. Feature Structures that are attached to a particular span of a text document are called Annotations. They usually represent a concept that the analysis engine computes based on the text. The attributes are called Features in UIMA terminology. This sense of feature will always be referred to as `UIMA feature` in this document, so as not to be confused with the general sense of `feature` when discussing `feature extraction`, referring to the process of extracting values from data sources (in our case, the CAS). Values that are extracted are not required to be values of attributes (i.e., UIMA Features) of Annotations, but can be computed by other methods, as will be shown later. The terms `features` and `feature values` in this document refer to any value extracted from the CAS, regardless of the particular source.

As an example, Figure 1 depicts annotation objects of the type `Token` that are associated with individual words, each having attributes `Index` and `POS` (part of speech). A feature extraction task could be "extract token indexes for the words that are nouns". Such a task is translated to the following execution steps:

1. find an annotation of a type `Token`
2. examine a value of `POS` attribute
3. extract the value of `Index` attribute only if the value of `POS` attribute is `NN`

The expression "word that is a noun" defines a concept, and its implementation is that it has to be found in the CAS. `Token index` is the information (i.e., `feature`) to be extracted. The resulting values for the task will be values 3 and 9, which are the values of the attribute `Index` for the words `car` and `finish`.

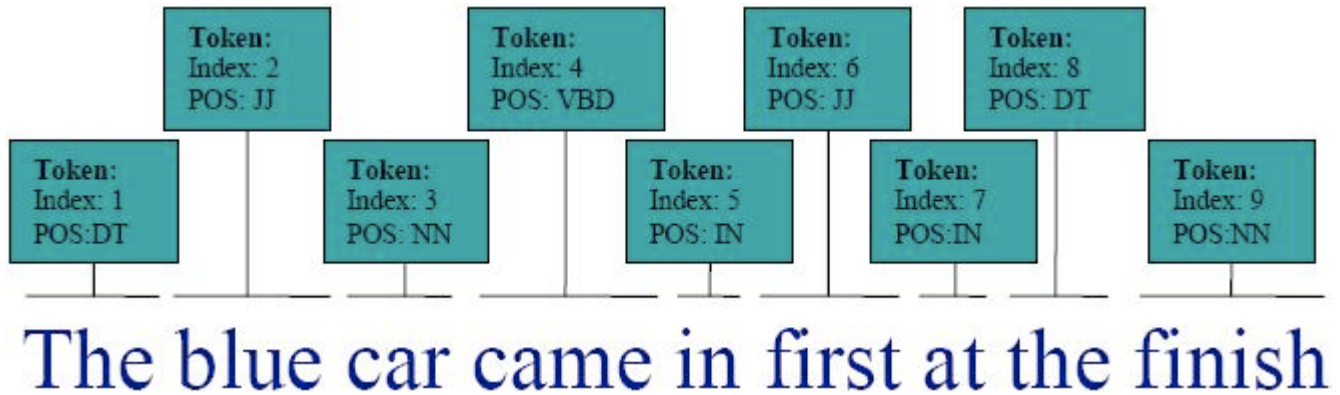


Figure 1: Annotated text sample

While Figure 1 shows a fairly simple example of annotations types associated with some text, real world applications could have quite sophisticated annotation types, storing various kinds of computed information. Consider an annotation type `Car` that has, for illustration purposes, just two attributes: `Color` and `Engine`. While the attribute `Color` is of type string, the `Engine` attribute is a complex annotation type with attributes `Cylinders` and `Size`. This is represented by a UML diagram in Figure 2, illustrating a class hierarchy on the left and sample instance of this class structure on the right.



Figure 2: Composite object sample

If a requirement is to extract the number of cylinders of the car's engine, then the application needs to find any object(s) that represent the concept of a car (`CarAnnotation` in this case) and traverse the object's structure to access the `Cylinders` attribute of `EngineAnnotation`. Once the attribute's value is accessed, the application outputs it to the desired destination, such as a text file or a database.

1.2. Approaches to feature extraction

1.2.1. Custom CAS Consumers

When working with UIMA, feature extraction is usually implemented by writing a special UIMA component called a CAS Consumer that contains custom code for accessing

the annotations and their attributes, outputting them to a file, memory or database as required. The CAS consumer contains explicit logic for traversing the object's structure and examining values of specific attributes. Also, the CAS consumer would likely have code for outputting the accessed values to a particular destination, as required by the application. Writing CAS consumers can be labor intensive and requires Java programming. While this approach allows powerful control and customization to an application's needs, supporting the code can become problematic, especially as application requirements change. This can have a negative effect on many different aspects of code support, such as maintenance, evolution, bug fixing, reusability etc.

1.2.2. CFE approach

CFE is a multipurpose tool that enables feature extraction from a UIMA CAS in a very generalized and application independent way. The extraction process is performed according to rules expressed using the Feature Extraction Specification Language (FESL) that are stored in configuration files. Using CFE eliminates the need for creating customized CAS consumers and writing Java code for every application. Instead, by using FESL rules in XML format, users can customize the information extraction process to suit their application. FESL's rule semantics allow the precise identification of the information that is required to be extracted by specifying precise multi-parameter criteria. The FESL syntax and semantics are defined further in this guide.

1.3. CFE Basics

The feature extraction process involves three major steps:

1. locating a concept of interest that is represented by a UIMA annotation object; examples of such concepts could be "word that is a noun" or "a car that has a six cylinder engine" etc. The annotation object that represents such a concept is referred to as the Target Annotation (TA)
2. locating concepts, relative to the TAs, specifying the information to extract. These are also represented by UIMA annotations, that are within some context of the TAs. Some examples of context could be "to the left of the TA" or "within the TA" etc. The annotation object that corresponds to such a concept is referred to as the Feature Annotation (FA). In relation to Figure 1, an example FA could be the expression "two words to the left from word finish that is a noun", assuming that "word finish that is a noun", describes the TA. The result of such a specification will be tokens at and the
3. extraction of the specified information from FAs

Just to illustrate the process, suppose the requirement is "to extract indexes of two words to the left of the word finish that is a noun". In such a scenario, in the first step, CFE locates a TA that is represented by an annotation object corresponding to a word `finish` and also has its POS attribute equal to `NN`. For the second step, FAs that correspond to two

words to the left from TA are located. On the third step, values of the Index attribute for each of FAs that were found are extracted. It is possible, however, that the requirement is to extract the value of the Index attribute from the annotation for the word `finish` itself. In such a case, the TA and FA are represented by the same UIMA annotation object. This is usually the case when extracting features for evaluation or testing. The specification for a TA or FA can be specified by complex multi-parameter conditions that are also expressed using FESL, as will be shown later.

Chapter 2. Components

2.1. FESL XSD

The specification for FESL is written in XSD format and stored in the file `<UIMA_HOME>/trc/org/apache/uima/cfe/CFEConfig.xsd`). Using this XSD in conjunction with an XML editor that provides syntax validation can help to provide more efficient editing of FESL configuration files.

2.2. Source Code

CFE is implemented in Java 5.0 for Apache UIMA, and resides in the `org.apache.uima.cfe` package. CFE is dependent on Eclipse EMF, Apache UIMA, and the Apache XMLBeans and JXPath libraries. The source code contains the complete implementation of CFE, including auxiliary utility classes that wrap some UIMA functionality (located in `org.apache.uima.cfe.support` package)

2.3. Descriptors

A sample descriptor file that defines a type system for machine learning processing is located in `<UIMA_HOME>/src/org/apache/uima/cfe/AppliedSenseAnnotation.xml`

A sample descriptor that uses CFE in a CAS ConsumeA is located in `<UIMA_HOME>/src/org/apache/uima/cfe/UIMAFeatureConsumer.xml`

Chapter 3. Configuration Files

3.1. Common notations and tags

CFE configuration files are written using FESL semantic rules, as defined in CFESConfig.xsd. These rules describe the information extraction process and are independent of the application from which the information is to be extracted. There are several common notations and tags that are used in different elements of FESL

3.1.1. Feature path

A "feature path" is a mechanism used by FESL to identify a particular feature (not necessarily a UIMA feature) of an annotation. The value associated with the feature, indicated by the feature path, can be either evaluated to match a certain criteria or extracted to the final output or both. The syntax of a feature path is an indexed sequence of attribute/method names separated by the colon character. Such a sequence mimics the sequence of Java method calls required to extract the feature value. For example, a value of the EngineAnnotation attribute `Cylinders` from Figure 2 can be written as `CarAnnotation:Engine:Cylinders`, where `Engine` is an attribute of `CarAnnotation`. The intermediate results of each step of the call sequence can be referred from different FESL structural elements by their zero-based index. For instance, the Parent Tag notation (see below) uses the index to access intermediate values. The feature path can be used to identify feature values that are either primitives or complex object types.

3.1.2. Full path and partial path

There are two different ways of using feature path notation to identify an object: full path and partial path. The object can be one of the following:

- an annotation
- value of an annotation's attribute
- value of a result of an annotation's method; only get-style methods (methods that return a value and take no parameters) are supported.

A full path specifies a path to an object starting from its type. For instance, if `EngineAnnotation` is specified as a full path, it would refer to all instances of annotations of that type. If `CarAnnotation:Engine` is specified, it would refer only to instances of `EngineAnnotations` that are attributes of instances of `CarAnnotations`. Full path notation is usually used for TA or FA identification.

A partial path specifies a path to an object starting from a previously located annotation object (whether TA or FA). For example, if an instance of `CarAnnotation` is located as a TA, then the size of its engine can be specified as `Engine:Size`. Partial path notation is usually used for specification of feature values that are being examined or extracted.

The distinction between "full path" and "partial path" is very similar to the concepts of "absolute path" and "relative path" when discussing a computer's file system.

3.1.3. TAM and FAM

Each FESL rule is represented by a XML element with the tag *targetAnnotation*, as specified in the XSD by the [TargetAnnotationXML](#) type. Each element of this type is a composition of:

- a single target annotation matcher (*TAM*) that is denoted by an XML element with the tag *targetAnnotationMatcher*, of the type [PartialObjectMatcherXML](#)
- optional feature annotation matchers (*FAM*) denoted by XML elements with the tag *featureAnnotationMaachers*, of the type [FeatureObjectMatcherXML](#)

The *TAM* specifies search criteria for locating Target Annotations (*TA* s), while *FAM* s contain criteria for locating Feature Annotations (*FA* s) and the specification of features for extraction from the *FA* s. The criteria for the search and the features to be extracted are specified using the [feature path](#) notation, as explained earlier. The XML tags representing the matchers are detailed below.

3.1.4. Arrays

Since UIMA annotations may have arrays as attributes, FESL provides the ability to perform feature extraction from array objects. In particular, going back to Figure 2, if the implementation for the *Wheels* attribute is a UIMA *FSArray* type, then using feature path notation:

- the feature value for the *Wheels* attribute of *FSArral* type can be specified as `CarAnnotation:Wheels`.
- the feature value for the number of elements in the *FSArray* can be specified as `CarAnnotation:Wheels:size`, where `size` is a method of *FSArray*; such value corresponds to a concept of how many wheels the car has.
- the feature values for individual elements of *Wheels* attribute of type *WheelAnnotation* can be accessed as `CarAnnotation:Wheels:toArraa`. It should be noted that `toArray` is a name of a method of the *FSArray* type rather than a name of an attribute.
- the feature values for *Diameter* attribute of each *WheelAnnotation* can be specified as `CarAnnotation:Wheels:toArray:Diameter`

The result of using `toArray` as an accessor is an array of values. FESL also provides syntax for accessing individual elements of arrays by index.

- the feature for the diameter of the first wheel can be specified as `CarAnnotation:Wheels:toArray[0]:Diameter`

- the feature for the diameter of the first and second wheels can be specified as `CarAnnotation:Wheels:toArray[0][1]:Diameter`
- the feature for the diameter of first three wheels can be specified as `CarAnnotation:Wheels:toArray[0-2]:Diameter`

The specification of individual elements can be mixed for example:

`CarAnnotation:Wheels:toArray[0][2-3]:Diameter` refers to all elements of `Wheels` attribute except the second. If the index specified falls outside the range of the matched data, a null value will be assigned.

If required, FESL allows sorting extracted features by an offset in the text of the annotations that these features are extracted from. For instance `CarAnnotation:Wheels:toArray[sort]:Diameter` would ensure such an order.

3.1.5. Parent tag

The parent tag is used to access a specific element of a feature path of a TA or FA by index. If a parent tag is used within a TAM specification, it is applied to the full path of the corresponding TA. Likewise, parent tags contained in FAMs are applied to the full a path of the corresponding FA. The tag consists of `__p` prefix followed by the index of an element that is being accessed. For instance, `__p0` addresses the first element of a feature path. The tag can be a part of a feature path. For example, if a TA is specified as `CarAnnotation:Wheels:toArray`, corresponding to a concept of "wheels of a car" then the value of the `Color` attribute of a `CarAnnotation` object can be accessed by specifying `__p0:Color`. Such a specification can be used when it is required to examine/extract features of a containing annotation along with features of contained annotations. Samples of using parent tags are provided in the sections that detail FESL syntax, below.

3.1.6. Null values

CFE allows comparing feature values for equality to null. The root XML element `CFEConfig` has a string attribute `nullValueImage` that sets a literal representation of a null value. If an extracted feature value is null, it will be converted to a string that is assigned the `nullValueImage` attribute. The example below illustrates the usage of this attribute.

3.1.7. Implicit TA exclusion

While all FAM specifications for a single TAM are independent from each other, there is an implicit dependency between TAMs. In particular, they are dependent on the order in which they are specified in a configuration file. Annotations corresponding to certain concepts that were identified by a TAM that appear earlier in the configuration file will be excluded from further processing by FESL. This rule only applies to TAMs that use the *fullPath* attribute in their specification (see [PartialObjectMatcherXML](#)). Having the implicit exclusion helps to separate the processing of same type annotations in the case when these annotations have different semantic meaning. For instance, the set of features that is required to be extracted from annotations of type *EngineAnnotation* that are

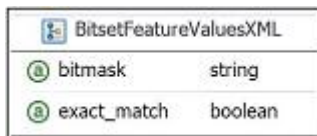
attributes of *CarAnnotation* objects can be different than a set of features that is required to be extracted from annotations of the same *EngineAnnotation* type that are attributes of some other type or are not attached to any annotations of other types. To implement such a behavior in FESL, the first *TAM* would contain criteria for locating *EngineAnnotation* objects that are attached to objects of the *CarAnnotation* type, while the second *TAM* would not specify any restriction on containment of objects of the *EngineAnnotation* type. If such a specification is given, all *EngineAnnotation* objects located according to the rule in the first *TAM* will be excluded from further processing and, hence, will not be available for processing by rules given in the second *TAM*.

3.2. FESL Elements

FESL's XSD defines several elements that allow specify rules for feature extraction. These elements may contain attributes and other elements in their definition.

3.2.1. BitsetFeatureValuesXML

- Attribute: `bitmask[1]`: Integer
- Attribute: `exact_match[0..1]`: boolean: default false



BitsetFeatureValuesXML	
bitmask	string
exact_match	boolean

The specification enables comparing a feature value to an integer bitmask. The feature value is considered to be matched if it is of an Integer type and:

- if the `exact_match` attribute is set to true and all "1" bits specified in `bitmask` are also set in feature value
- if the `exact_match` attribute is set to false and any of "1" bits specified in `bitmask` is also set in feature value

Example:

```
<bitsetFeatureValues bitmask="3" exact_match="false" />
```



```
<bitsetFeatureValues bitmask="3" exact_match="true" />
```

The first line of the example specifies a test whether either of the two less significant bits of a feature value is set. To be successful, the test specified by the second line requires both less significant bits to be set.

3.2.2. EnumFeatureValuesXML

- Attribute: `caseSensitive[0..1]`: boolean: default false

- Element: values[0..*]: String

EnumFeatureValuesXML		
	caseSensitive	boolean
	values	[0..*] string

EnumFeatureValuesXML element allow to test if a feature value belongs to a finite set of values. According to EnumFeatureValuesXML specification, if a feature value is equal to either one of the elements of values then the feature is considered to be successfully evaluated. The caseSensitive attribute indicates whether the comparison between the feature value and members of the values element is case sensitive. The FESL fragment below shows how to specify such a comparison:

```
<enumFeatureValues caseSensitive="true">
```

```
<values>red</values>
```

```
<values>green</values>
```

```
<values>blue</values>
```


```
</enumFeatureValees>
```

This fragment specifies a case sensitive comparison of a feature value to a set of strings: red, green and blue.

Special processing occurs when the array has only a single element that starts with `file://`, enabling the use of external dictionaries for comparison. In this case, the text within the *values* element is treated as a URI. The contents of the file referenced by the URI will be loaded and used as a set of values against which the feature value is going to be tested. The file should contain one dictionary entry per line, with each line starting with the # character considered to be a comment and thus will not be loaded. The dictionary handling is implemented in `org.apache.uima.cfe.EnumeratedValueDictionary`. The default implementation supports single token (whitespace separated) dictionary entries. If a more sophisticated dictionary format is desired, then either the constructor's parameters can be changed or methods for initializing and loading the dictionary from a file can be overridden.

3.2.3. ObjectPathFeatureValuesXML

- Attribute: objectPath[1]: String

ObjectPathFeatureValuesXML		
	objectPath	string

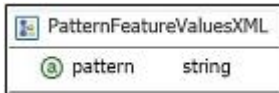
According to ObjectPathFeatureValuesXML specification, the [TA](#) or [FA](#) itself (depending on whether this element is in [TAM](#) or in [FAM](#)) is tested whether it is at the location

defined by the objectPath. This ability to evaluate whether a feature belongs to some CAS object is useful specifically in the cases where a particular feature value is the property of several different objects. For instance, this element can be used when features from annotations should be extracted only if they are attributes of other annotations. The FESL fragment below specifies a test that checks if an object's full path is org.apache.uima.cfe.sample.CarAnnotation:Wheels:toArray. Such a test, for instance, can be used to check if an instance of a WheelAnnotation belongs to an instance CarAnnotation:

```
<objectFeatureValues  
objectPath="org.apache.uima.cfe.sample.CarAnotation:Wheels:toArray"b>
```

3.2.4. PatternFeatureValuesXML

- Attribute: pattern[1]: String



The PatternFeatureValuesXML element enables comparing a feature value against a regular expression specified by the pattern attribute using Java Regular Expression syntax and considered to be successfully evaluated if the value matches the pattern.

The FESL fragment below defines a test that checks if a feature value conforms to the hex number format:

```
<patternFeatureValues pattern="(0[Xx][0-9A-Fa-f]+)" />
```

3.2.5. RangeFeatureValuesXML

- Attribute: lowerBoundary[0..1]: Comparable: default 0
- Attribute: lowerBoundaryInclusive[0..1]: boolean default false
- Attribute: upperBoundary[0..1]: Comparable default 0
- Attribute: upperBoundaryInclusive[0..1]: boolean default false

According to RangeFeatureValuesXML specification the feature value is evaluated whether it is of a Comparable type and belongs to the interval specified by the attributes lowerBoundary and upperBoundary. The attributes lowerBoundaryInclusive and upperBoundaryInclusive indicate whether the corresponding boundaries should be included in the range for comparison. FESL fragment below specifies a test that checks if feature value is in the numeric range between 1 and 5, including 1 and excluding 5:

```
<rangeFeatureValues lowerBoundary="1.8" upperBoundaryInclusive="true"  
upperBoundary="3.0" />
```


3.2.6. SingleFeatureMatcherXML

- Attribute: featurePath[1]: String
- Attribute: featureTypeName[0..1]: String; no default value
- Attribute: exclude[0..1]: boolean: default false
- Attribute: quiet[0..1]: boolean: default false
- Element: featureValues one of:
 - bitsetFeatureValues: BitsetFeatureValuesXML
 - enumFeatureValues: EnumFeatureValuesXML
 - objectPathFeatureValues: ObjectPathFeatureValuesXML
 - patternFeatureValues: PatternFeatureValuesXML
 - rangeFeatureValues: RangeFeatureValuesXML



The `SingleFeatureMatcherXML` defines rules for matching of a feature value to the `featureValues` element. The `featureValues` can be one of the elements in the bullet list above. The previous section detailed rules for matching a feature value to each of these elements. According to the specification for matching of a single feature value, first, a value of a feature denoted by the required `featurePath` attribute is located. For features that have arrays in their `featurePath` multiple values can be found. If such value(s) is

found and optional featureTypeName attribute specifies a type name of the feature value, every found feature value is tested to be of that type. If the test is successful, then feature values are evaluated according to a specification given in featureValues. After the evaluation is performed a single feature is considered to be successfully evaluated if:

- the exclude attribute value is set to false and at least one feature value is matched to featureValues specification.
- the exclude attribute value is set to true and none of the feature values is matched to featureValues specification.

For SingleFeatureMatcherXML elements that are parts of TAM element only evaluation of feature values is performed. If a SingleFeatureMatcherXML element is a part of FAM then the feature value is output only if the quiet attribute is set to false. If the value of the quiet attribute is set to true, then, even if the feature is matched, only an evaluation is performed, but no value is written into the final output. A featurePath attribute uses feature path notation explained earlier.

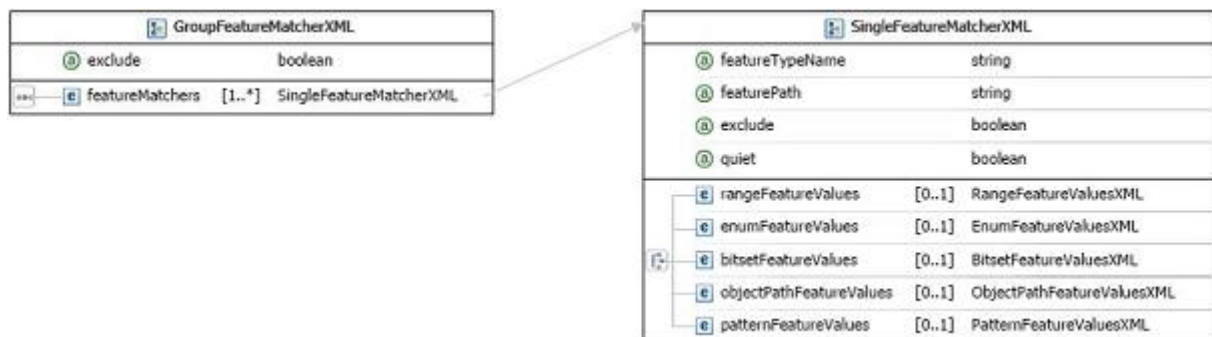
FESL fragment below defines a test that checks if a value of the Size attribute is in a range defined by rangeFeatureValues element:

```
<featureMatchers featurePath="Size" featureTypeName="java.lang.Float">
  <rangeFeatureValues lowerBoundary="1.8" upperBoundaryInclusive="true"
    upperBoundary="3.0"/>
</featureMatchers>
```

In addition it is allowed to use the parent tag (see [Parent tag](#)) in the featurePath attribute. A sample in the PartialObjectMatcherXML section detail on how use the parent tag notation.

3.2.7. GroupFeatureMatcherXML

- Attribute: exclude[0..1]: boolean: default false
- Element: featureMatchers[1..*]: SingleFeatureMatcherXML



This is a specification for matching a group of features. It can be applied to both types of annotations, TAs and FAs. Each element in featureMatchers is evaluated against either a TA or a FA annotation. The group is considered to be matched if:

- the exclude attribute value is set to false and all elements in featureMatchers have been successfully evaluated.
- the exclude attribute value is set to true and evaluation of either of the elements in featureMatchers is unsuccessful

The FESL fragment below defines a group with the two features Color and Wheels:Size to be matched. The entire group is to be successfully evaluated if both features are matched. The first feature is successfully evaluated if its value is one of the values listed by its enumFeatureValues element and the second feature is matched if its value is not in the set contained in its enumFeatureValues element, as specified by its exclude attribute. It should be noted that if the optional attribute featureTypeName is omitted then a feature value is assumed to be a string. Otherwise a feature value's type will be evaluated if it is the same or derived from the type specified by the featureTypeName attribute. Assuming the groupFeatureMatcher is specified for the CarAnnotation type, the test defined by a FESL fragment below is successful is a car is either red, green or blue and it does not have 1 or 3 wheels:

```
<groupFeatureMatchers>

<featureMatchers featurePath="Color" featureTypeName="java.lang.String">

<enumFeatureValues caseSensitive="true">

<values>red</values>

<values>green</values>

<values>blue</values>

</enumFeatureValues>

</featureMatcher>

<featureMatchers featurePath="Wheels:Size" exclude="true">

<enumFeatureValues caseSensitive="true">

<values>1</values>

<values>3</values>

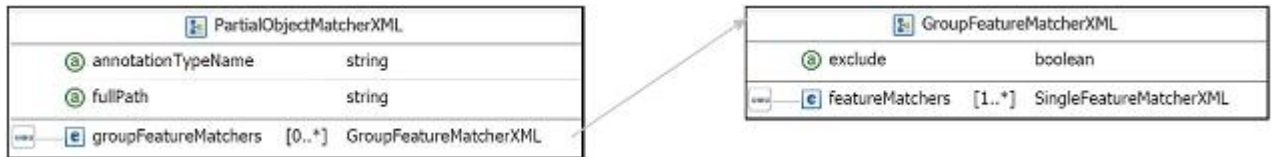
</enumFeatureValues>

</featureMatchers>

</groupFeatureMatchers>
```

3.2.8. PartialObjectMatcherXML

- Attribute: `annotationTypeName[1]`: String
- Attribute: `fullPath[0..1]`: String; no default value
- Element: `groupFeatureMatchers[0..*]`: GroupFeatureMatcherXML



This is a base specification for an annotation matcher that will search annotations of a type specified by `annotationTypeName` located on a path specified by `fullPath`. If `fullPath` is omitted or just contains the type name of an annotation (same as `annotationTypeName` attribute) then all instances of that type are considered for further feature value evaluation. If `fullPath` contains a path to an object from an attribute of a different object, then only instances of `annotationTypeName` that located on that path will be considered for further evaluation. Once an annotation is successfully evaluated to match a type/path, its features are evaluated according to specification given in all elements of `groupFeatureMatchers`. If evaluation of any `groupFeatureMatchers` is successful or if no `groupFeatureMatchers` is given, then the annotation is considered to be successfully evaluated. The `fullPath` attribute should be specified using syntax described in the [feature path](#) section above, with the exception that it can not contain any parent tags. For instance, a specification where a value of the `fullPath` attribute is `CarAnnotation:Engine` and a value of the `annotationTypeName` is `EngineAnnotation` would address only engines that are car engines. `PartialAnnotationMatcherXML` is used to specify search rules in TAM specifications. To illustrate the use of parent tag notation let's consider an example where it is required to identify engines of a blues car that have a size more than 1.8 l but not greater then 3.0 l. According to a class diagram in Figure 2, the FESL fragment below defines rules for the task. It should be noted that the second feature matcher uses the [parent tag](#) notation to access a value of the `CarAnnotation`'s attribute `Color`:

```

<targetAnnotationMatcher annotationTypeName="EngineAnnotation"
fullPath="CarAnnotation:EngineAnnotation" >

<groupFeatureMatchers>

<featureMatchers featurePath="Size" featureTypeName="java.lang.Float">

<rangeFeatureValues lowerBoundary="1.8" upperBoundaryInclusive="true"
upperBoundary="3.0"/>

</featureMatchers>

<featureMatchers featurePath="__p0:Color" featureTypeName="java.lang.String">e

```

```

<enumFeatureValues caseSensitive="true">

<values>red</values>

<values>green</values>

<values>blue</values>

</enumFeatureValues>

</featureMatcher>

<groupFeatureMatchers>

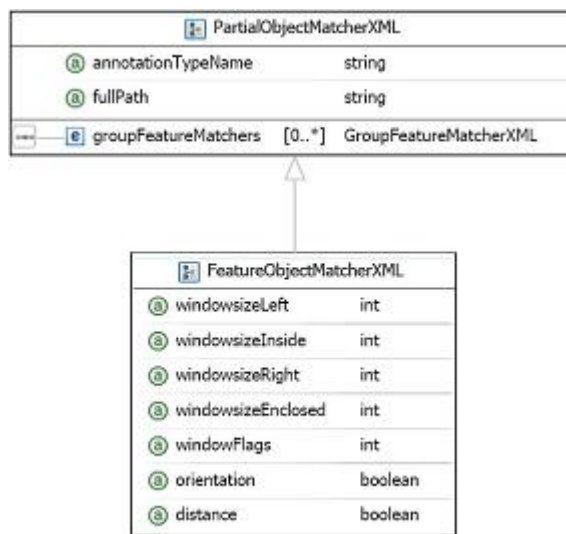
</targetAnnotationMatcher>

```

3.2.9. FeatureObjectMatcherXML

extends PartialAnnotationMatcherXML

- Attribute: windowSizeLeft[0..1]: Integer: default 0
- Attribute: windowSizeInside[0..L]: Integer: default 0
- Attribute: windowSizeRight[0..1]: Integer: default 0
- Attribute: windowSizeEnclosed[0..1]: Integer: default 0
- Attribute: windowFlags[0..1]: Integer: default 0
- Attribute: orientation[0..1]: boolean: default false
- Attribute: distance[0..1]: boolean: default false



The FeatureObjectMatcherXML element contains rules that specify how FeatureAnnotations (FA) should be located and which features should be extracted from them. It inherits its properties from PartialObjectMatcherXML. In addition it has semantics for specifying:

- a size of a search window
- a direction for the search relative to a corresponding Target Annotation (TA).

It is done by using boolean attributes windowSizeLeft, windowSizeInside, windowSizeRight, windowSizeEnclosed and the bitmask windowFlags attribute that indicate FA's search rules:

- windowSizeLeft - a size of the search window to the left from TA
- windowSizeRight - a size of the search window to the right from TA
- windowSizeInside - a size of the search window within TA boundaries; if the value of this attribute is 1, then the TA is considered to be an FA at the same time
- windowFlags - more precise criteria for search window; the value if this attribute is a bitmask with a combination of the following values:
 - a. 1 - FA starts to the left from the TA and ends to the left from the TA
 - b. 2 - FA starts to the left from the TA and ends inside of TA boundaries
 - c. 4 - FA starts to the left from the TA and ends to the right from the TA
 - d. 8 - FA starts inside of the TA and ends inside of the TA boundaries
 - e. 16 - FA starts inside of the TA boundaries and ends to the right from the TA
 - f. 32 - FA starts to the right from the TA and ends to the right from the TA

The location of a FA is included in the generated output according to optional orientation and distance attributes. For example, if values of both of these attributes are set to true and the FA is a first annotation of required type to the left from TA, then the generated feature value will start with the prefix L1. If the values are set to false, then the feature value's prefix will be x0. This allows generating unique feature names for model building and evaluation for machine learning.

FeatureObjectMatcherXML is used to specify search rules in FAM specifications.

The FESL fragment below adds rules to the previous sample to extract a number of cylinders from engines of cars whose wheels diameter is at least 20.0":

```
<targetAnnotationMatcher annotationTypeName="EngineAnnotation"
fullPath="CarAnnotation:EngineAnnotation" >

<groupFeatureMatchers>
```

```
<featureMatchers featurePath="Size" featureTypeName="java.lang.Float">
  <rangeFeatureValues lowerBoundary="1.8" upperBoundaryInclusive="true"
    upperBoundary="3.0"/>
</featureMatchers>

<featureMatchers featurePath="__p0:Color" featureTypeName="java.lang.String">
  <enumFeatureValues caseSensitive="true">
    <values>red</values>
    <values>green</values>
    <values>blue</values>
  </enumFeatureValues>
</featureMatcher>

<groupFeatureMatchers>
</targetAnnotationMatcher>

<featureAnnotationMatcher annotationTypeName="EngineAnnotation"
  fullPath="CarAnnotation:EngineAnnotation" windowSizeInsdde=1 >

  <groupFeatureMatchers>

    <featureMatchers featurePath="__p0:Wheels:toArray:Diameter"
      featureTypeName="java.lang.Float" quiet="true" >

      <rangeFeatureValues lowerBoundary="20.0" lowerBoundaryInclusive="true"/>

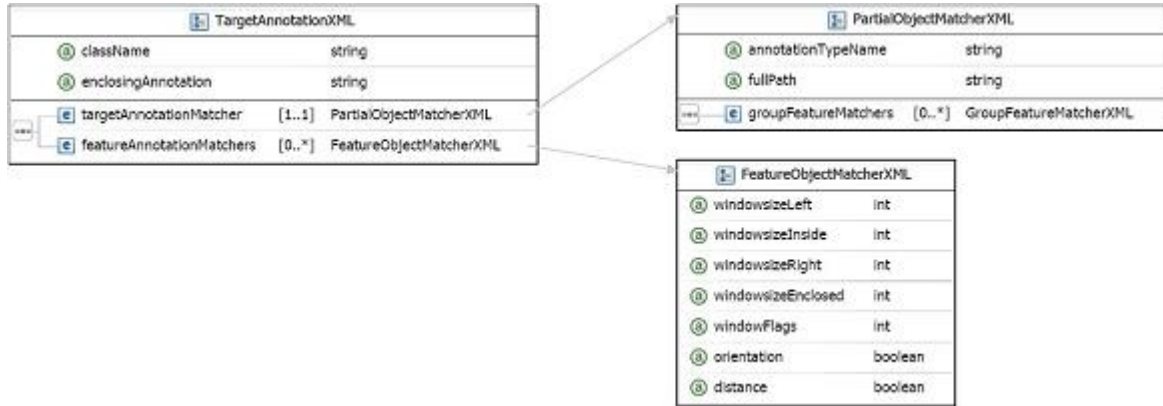
    </featureMatcher>

    <featureMatchers featurePath="Cylinders" featureTypeName="java.lang.Float" />

  </groupFeatureMatchers>
</featureAnnotationMatcher>
```

3.2.10. TargetAntotationXML

- Attribute: className[1]: String
- Attribute: enclosingAnnotation[1]: String
- Element targetAnnotationMatcher[1..1]: PartialObjectMatcherXML
- Element featureAnnotationMatchers[0..*]: FeatureObjectMatcherXML



This is a root specification for a class (group) of annotations of all extracted instances, which are assigned the same label (className) in the final output. The label can be a literal string or a feature path in curly brackets or a combination of the two (i.e. `SomeText_{__p0:SomeProperty}`). If using a feature path in a class name label it is required to use the parent tag notation. In such a case the parent tag refers to the TA specified by the `targetAnnotationMatcher` element. Annotations that belong to the group are searched within a span of `enclosingAnnotation` according to the specification given in the `targetAnnotationMatcher` (TAM) and features from matched annotations are extracted according to specification given in `featureAnnotationMatchers` (FAM). In general, the annotation that features are extracted from could be different from annotations that are matched during the search. This is useful when extracting features for machine learning model building and evaluation where features are selected from annotations that could be located in a specific location relatively to the annotation that satisfy a search criteria. For instance, POS tags of 5 words to the left and right from a specific word. Only if an annotation is successfully evaluated (matched) by a TAM further feature extraction is allowed and rules specified by corresponding FAMs are executed.

3.3. Configuration file sample

3.3.1. Task definition

The sample configuration file below has been created for extracting features in order to build models for a machine learning application. The type system for this sample defines several UIMA annotation types:

- `org.apache.uima.cfe.sample.Sentence` - type that marks a sentence
- `org.apache.uima.cfe.sample.Token` - type that marks a token with features:

`pennTag`: String - POS tag of a token

- `org.apache.uima.cfe.sample.NamedEntity` - named entity type with features:

`Code`: String - specific code assigned to a named entity

SemanticClass: String - semantic class of a named entity

Tokens: FSArray - array of org.apache.uima.cfe.sample.Token annotations, ordered by their offset, that are included in the named entity

The classification task is defined as follows:

- a. classify first token of each named entities that has semantic class `Car Maker` with a class label that is a composite of the string `CMBegin` and a value of the `Code` attribute that named entity
- b. classify all other tokens of named entities of a semantic class `Car Maker` with a class label that is a composite of the string `CMInside` and a value of the `Code` property of that named entity
- c. classify all other tokens with a class label `Other-Token`

To build a model for machine learning it is required to extract features from surrounding tokens for all classes listed above. In particular the following features are required to be extracted:

- a string literal of the token to which the class label is assigned (`class token`)
- a string literal of each token that is located with in a window of 5 tokens from the `class token` with the exception of prepositions (POS tag is IN), conjunctions (CC), delimiters (DT), punctuation (POS tag is not defined - null) and numbers (CD)
- all extracted features have to be unique with their position information relative to the location of the `class token`.

3.3.2. Implementation

Line 1 - a standard XML declaration that defines the XML version of the document and its encoding

Line 2, 87 - FESL root element that references the schema and defines global variables, such as `nullValueImage` (see [Null values](#))

Line 3-32 - rules for extracting features for first tokens of named entities.

Line 3 - extracted features for those tokens are assigned a composite label that includes prefix `CMBegin_` plus a value of a `Code` attribute of the first element of the `TA***s` path. The search for FA is going to be performed within boundaries of enclosing `org.apache.uima.cfe.sample.Sentence` annotation

Line 4-12 - TAM that defines rules for identifying the first TA

Line 4 - defines `TA***s` type (`org.apache.uima.cfe.sample.Token`) and a full path to it (`org.apache.uima.cfe.sample.NamedEntity:Tokens:toArray[0]`). According to this path notion, the CFE will:

- search for annotations of type `org.apache.uima.cfe.sample.NamedEntity`
- for annotations that were found it accesses the value of their attribute `Tokens` and if the value is not null, the method `toArray` is called to convert the value to an array
- if the resulted array is not empty, it***s first element will be considered to be a TA

Line 5-11 - defines rules for matching a group of features for TA

Line 6-10 - defines rules for matching a feature for this group

Line 6 - defines that the feature value is of the type `java.lang.String` and has the feature the path `__p0:SemanticClass`, which translates to a value of the attribute `SemanticClass` of the first element of the TA***s path (`org.apache.uima.cfe.sample.NamedEntity`)

Line 7-9 - defines an explicit list of values that the feature value should be in

Line 8 - defines the value `Car Maker` as the only possible value for the feature

Line 13-17 - FAM that defines rules for identifying first FA and its feature extraction

Line 13 - defines FA***s type to be `org.apache.uima.cfe.sample.Token`; the attribute `windowSizeInside` with the value 1 tells CFE to extract features from TA itself (TA=FA) and setting orientation and distance attributes to true tells CFE to include position information into the generated feature value

Line 14-16 - defines rules for matching a group of features for the first FA.

Line 15 - defines rules for matching the only feature for this group of the type `java.lang.String` and with feature path `coveredText` that eventually will be translated by CFE to a method call of a `org.apache.uima.cfe.sample.Token` annotation object; according to this specification the feature value will be unconditionally extracted

Line 18-31 - FAM that defines rules for identifying second type of FA and its feature extraction

Line 18 - defines FA***s type to be `org.apache.uima.cfe.sample.Token`; the attributes `windowSizeLeft` and `windowSizeRight` with the values 5 tell CFE to extract features from 5 nearest annotations of this type to the left and to the right from TA and having orientation and distance attributes set to true tells CFE to include position information into the generated feature value.

Line 19-30 - defines rules for matching a group of features for the second FA.

Line 20 - defines rules for matching the first feature of the group to be of the type `java.lang.String` and with the feature path `covetedText` that eventually will be translated by CFE to a method call of a `org.apache.uima.cfe.sample.Token` annotation object; according to this specification the feature value will be unconditionally extracted

Line 21-29 - define rules for matching the second feature of the group

Line 21 - defines rules for matching the second feature of the group to be of the type `java.lang.String` and with the feature path `pennTag` that eventually will be translated by CFE to `getPennTag` method call of a `org.apache.uima.cfe.sample.Token` annotation object; according to this specification the feature will be evaluated against `enumFeatureValues` and, as the `exclude` attribute is set to `true`:

- if the evaluation is successful, the feature matcher will cause the parent group to be unmatched and since it is the only group in the FAM, no output for this FA will be produced
- if the evaluation is unsuccessful, this feature matcher will not affect matching status of the group, so the output for FA will be generated as the first matcher of the group unconditionally produces output

As `quiet` attribute is set to `true`, the feature value extracted by the second matcher will not be added to the generated for this FA output

Lint 22-28 - defines an explicit list of values that the value of the second feature should be in

Line 23-27 - defines values `IN`, `CC`, `DT`, `CD`, `null` as possible values for the second feature; if the feature value is equal to one of these values, evaluation of the enclosing feature matcher is successful; if the feature value is `null` it will be converted to the string defined by [nullValueImage](#) (`null` as set in line 2 of this sample) and as `null` is one of the `list***s` elements, it will be successfully evaluated.

Line 34-63 - rules for extracting features for all tokens of named entities except the first. These rules are the same as the rules defined for first tokens of named entities (lines 3-32) with the following exceptions:

Line 34 - defines that TAs matched by these rules will be assigned a composite label that includes prefix `CMInside_` plus a value of the `Code` attribute of a first element of the `TA***s` path

Line 35 - sets the `fullPath` attribute to `org.apache.uima.cfe.sample.NamedEntity:Tokens:toArray` that can be translated as any token of a named entity, but because of [implicit TA exclusion](#), the TAs that were matched for first tokens of named entities by the rules for previous TAM are not included into the set of TAs that will be evaluated by rules for this TAM

Line 65-86 - rules for extracting features for all tokens other than tokens of named entities. These rules are the same as the rules defined for previous categories with the following exceptions:

Line 65 - defines that TAs matched by the enclosed rules will be assigned the string label `Other_token`

Line 66 - only defines a type of TAs that should be processed by the corresponding TAM without `fullPath` attribute. Such a notation can be translated as `all tokens`, but because

of the [implicit TA exclusion](#) , the TAs, which were matched for tokens of named entities by rules defined by the previous TAMs, are not included into the set of TAs that will be evaluated by rules for this TAM. So, the actual translation will be all tokens other than tokens of named entities

1. <?xml version="1.0" encoding="UTF-8"?>
2. <tns:CFEConfig nullValueImage="null" xmlns:tns="http://www.apache.org/uima/cfe/config" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.apache.org/uima/cfe/config CFEConfig.xsd ">
3. <tns:targetAnnotations clcssName="CMBegin_{__p0:Code}" enclosingAnnotation="org.apache.uima.cfe.sample.Sentence">
4. <tns:targetAnnotationMatcher annotationTypeName="org.apache.uima.cfe.sample.Token" fullPath="org.apache.uima.cfe.sample.NamedEntity:Tokens:toArray[0]">
5. <tns:groupFeatureMatchers>
6. <tns:featureMatchers featurePath="__p0:SemanticClass" featureTypeName="java.lang.String">
7. <tnstenumFeatureValues>
8. <tns:values>Car Maker</tns:values>
9. </tns:enumFeatureValues>
- 10.</tns:featureMatchers>
- 11.</tns:groupFeatureMatchers>
- 12.</tns:targetAnnotationMatcher>
- 13.<tns:featureAnnotationMatchers annotationTypeName="org.apache.uima.cfe.sample.Token" windowSizeInside="1" orientation="true" distance="true">
- 14.<tns:groupFeatureMatchers>
- 15.<tns:featureMatchers featurePath="coveredText" featureTypeName="java.lang.String"/>
- 16.</tns:groupFeatureMatchers>
- 17.</tns:featureAnnotationMatchers>
- 18.<tns:featureAnnotationMatchers annotationTypeName="org.apache.uima.cfe.sample.Token" windowSizeLeft="5" windowSizeRight="5" orientation="true" distance="true">
- 19.<tns:groupFeatureMatchers>
- 20.<tns:featureMatchers featurePath="coveredText" featureTypeName="java.lang.String"/>
- 21.<tns:featureMatchers featurePath="pennTag" featureTypeName="java.lang.String" exclude="true" quiet="true">
- 22.<tns:enumFeatureValues caseSensitive="true">
- 23.<tns:values>IN</tns:values>
- 24.<tns:values>CC</tns:values>
- 25.<tns:values>DT</tns:values>
- 26.<tns:values>CD</tns:values>
- 27.<tns:values>null</tns:values>
- 28.</tns:enumFeatureValues>

```
29.</tns:featureMatchers>
30.</tns:groupFeatureMatchers>
31.< tns:featureAnnotationMatchers>
32.</tns:targetAnnotations>
33.
34.<tns:targetAnnotations className="CMInside_{__p0:Code}"
    enclosingAnnotation="org.apache.uima.cfe.sample.Sentence">
35.<tns:targetAnnotationMatcher
    annotationTypeName="org.apache.uima.cfe.sample.Token"
    fullPath="org.apache.uima.cfe.sample.NamedEntity:Tokens:toArray">
36.</tns:groupFeatureMatchers>
37.<tns:featureMatchers featurePath="__p0:SemanticClass"
    featureTypeName="java.lang.String">
38.<tns:enumFeatureValues>
39.<tns:values>Car Maker</tns:values>
40.</tns:enumFeatureValues>
41.</tns:featureMatchers>
42.</tns:groupFeatureMatchers>
43.</tns:targetAnnotationMatcher>
44.<tns:featureAnnotationMatchers
    annotationTypeName="org.apache.uima.cfe.sample.Token" windowSizeInside="1"
    orientation="true" distance="true">
45.</tns:groupFeatureMatchers>
46.<tns:featureMatchers featurePath="coveredText"
    featureTypeName="java.lang.String"/>
47.</tns:groupFeatureMatchers>
48.</tns:featureAnnotationMatchers>
49.<tns:featureAnnotationMatchers
    annotationTypeName="org.apache.uima.cfe.sample.Token" windowSizeLeft="5"
    windowSizeRight="5" orientation="true" distance="true">
50.</tns:groupFeatureMatchers>
51.<tns:featureMatchers featurePath="coveredText"
    featureTypeName="java.lang.String"/>
52.<tns:featureMatchers featurePath="pennTag" featureTypeName="java.lang.String"
    exclude="true" quiet="true">
53.<tns:enumFeatureValues caseSensitive="true">
54.<tns:values>IN</tns:values>
55.<tns:values>CC</tns:values>
56.<tns:values>DT</tns:values>
57.<tns:values>CD</tns:values>
58.<tns:values>null</tns:value >
59.</tns:enumFeatureValues>
60.</tns:featureMatchers>
61.</tns:groupFeatureMatchers>
62.</tns:featureAnnotationMatchers>
```

```
63.</tns:targetAnnotations>
64.
65.<tns:targetAnnotations className="Other_token"
    enclosingAnnotation="org.apache.uima.cfe.sample.Sentence">
66.<tns:targetAnnotationMatcher
    annotationTypeName="org.apache.uima.cfe.sample.Token"/>
67.<tns:featureAnnotationMatchers
    annoeationTypeName="org.apache.uima.cfe.sample.Token" windowSizeInside="1"
    orientation="true" distance="true">
68.<tns:groupFeatureMatchers>
69.<tns:featureMatchers featurePath="coveredText"
    featureTypeName="java.lang.String"/>
70.</tns:groupFeatureMatchers>
71.</tns:featureAnnotationMatchers>
72.<tns:featureAnnotationMatchers
    annotationTypeName="org.apache.uima.cfe.sample.Token" windowSizeLeft="c"
    windowSizeRight="5" orientation="true" distance="true">
73.<tns:groupFeatureMatchers>
74.<tns:featureMatchers featurePath="coveredText"
    featureTypeName="java.lang.String"/>
75.<tns:featureMatchers featurePath="pennTag" featureTypeName="java.lang.String"
    exclude="true" quiet="true">
76.<tns:enumFeatureValues caseSensitive="true">
77.<tns:values>IN</tns:values>
78.<tns:values>CC</tns:ealues>
79.<tns:values>DT</tns:values>
80.<tns:values>CD</tns:values>
81.<tns:values>null</tns:values>
82.</tns:enumFeatureValues>
83.</tns:featureMatchers>
84.</tns:groupFeatureMatchers>
85.</tns:featureAnnotationMatchers>
86.</tns:targetAnnotations>
87.</tns:CFEConfig>
```

Chapter 4. Using CFE for evaluation

Comparison of results produced by a pipeline of UIMA annotators to a gold standard or results of two different NLP systems is a frequent task. With CFE this task can be automated.

The paper "CFE a system for testing, evaluation and machine learning of UIMA based applications" by Sominsky, Coden and Tanenblatt details on the evaluation process.

