

Apache UIMA ConceptMapper Annotator Documentation

**Written and maintained by the Apache
UIMA™ Development Community**

Version 2.10.2

Copyright © 2006, 2018 The Apache Software Foundation

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.

Publication date April, 2018

Table of Contents

Introduction	v
1. Using ConceptMapper	1
2. Functionality	3
2.1. Dictionaries	3
2.2. Dictionary Entry Tokenization	4
2.3. Input Document Processing	4
2.4. Lookup Strategies	5
2.5. Output Options	6
3. Configuration Parameters	9

Introduction

ConceptMapper is a highly configurable, high performance dictionary lookup tool, implemented as a UIMA (Unstructured Information Management Architecture) component. Using one of several matching algorithms, it maps entries in a dictionary onto input documents, producing UIMA annotations.

Chapter 1. Using ConceptMapper

ConceptMapper was designed to provide highly accurate mappings of text into controlled vocabularies, specified as dictionaries, including the association of any necessary properties from the controlled vocabulary as part of that mapping. Individual dictionary entries could contain multiple terms (tokens), and ConceptMapper can be configured to allow multi-term entries to be matched against non-contiguous text. It was also designed to perform fast, and has been easily able to provide real-time results even with multi-million entry dictionaries.

Lookups are token-based, and are limited to applying within a specific context, usually a sentence, though this is configurable (e.g., a noun phrase, a paragraph or other NLP-based concept).

Chapter 2. Functionality

There are many parameters to configure all aspects of ConceptMapper's functionality, in terms of:

- processing the dictionary
- the way input documents are processed
- the availability of multiple lookup strategies
- its various output options

2.1. Dictionaries

The requirements on the design of the ConceptMapper dictionary were that it be easily extensible and that arbitrary properties could be associated with individual entries. Additionally, the set of properties could not be fixed, but rather customizable for any particular application.

The structure of a ConceptMapper dictionary is quite flexible and is expressed using XML (see [Example 2.1, “Sample dictionary entry” \[3\]](#)). Specifically, it consists of a set of entries, specified by the <token> XML tag, each containing one or more variants (synonyms), the text of which is specified using by the "base" feature of the <variant> XML tag. Entries can have any number of associated properties, as needed. Individual variants (synonyms) inherit features from their parent token (i.e., the canonical form), but can override any or all of them, or even add additional features.

In the following sample dictionary entry, there are 6 variants, and according to the rules described earlier, each inherits the all attributes from the canonical form (canonical, CodeType, CodeValue, and SemClass), though the variants “colonic” and “colic” override the value of the POS (part of speech) attribute:

```
<token canonical="colon, nos"
      CodeType="ICDO" CodeValue="C18.9"
      SemClass="Site" POS="NN">
  <variant base="colon, nos" />
  <variant base="colon" />
  <variant base="colonic" POS="JJ" />
  <variant base="colic" POS="JJ" />
  <variant base="large intestine" />
  <variant base="large bowel" />
</token>
```

Example 2.1. Sample dictionary entry

The result of running ConceptMapper are UIMA annotations, and there are two configuration parameters that are used to map the attributes from the dictionary (see [AttributeList \[9\]](#)) to features of UIMA annotations (see [FeatureList \[9\]](#)).

The entire dictionary is loaded into memory, which, in conjunction with an efficient data structure, provides very fast lookups. As stated earlier, dictionaries with millions of entries have been used without any performance issues. The obvious drawback to storing the dictionary in memory is that large dictionaries require large amounts of memory; this is partially mitigated by the fact that the dictionary is implemented as a UIMA shared resource (see [DictionaryFile \[11\]](#)). This

means that multiple annotators, such as multiple instances of `ConceptMapper` that are set up using different parameters, can all access it without having to load it more than once.

The dictionary loader is specified in the external resource section of the descriptor, and is expected to implement the interface `org.apache.uima.conceptMapper.support.dictionaryResource.DictionaryResource`. Two implementations are included in the distribution.

- `org.apache.uima.conceptMapper.support.dictionaryResource.DictionaryResource_impl`, the standard implementation, which loads an XML version of a dictionary.
- `org.apache.uima.conceptMapper.support.dictionaryResource.CompiledDictionaryResource_impl`, which loads a pre-compiled version, for faster loading.

The compiler is supplied as `org.apache.uima.conceptMapper.dictionaryCompiler.CompileDictionary`, which takes two arguments, a `ConceptMapper` analysis engine descriptor that loads the dictionary using the standard dictionary loader, and the name of the output file into which to write the compiled dictionary.

A compiled version of the dictionary can optionally be created while loading the non-compiled version, by specifying the Java property `uima.conceptmapper.compiled_dictionary_directory`, giving it the value of a path to a writable directory.

2.2. Dictionary Entry Tokenization

Input documents are processed on a token-by-token basis, so it is important that the dictionary entries are tokenized in the same way as the input documents. To accomplish this, `ConceptMapper` allows any UIMA analysis engine to be specified as the tokenizer for the dictionary entries. See parameter [TokenizerDescriptorPath \[9\]](#) for details.

2.3. Input Document Processing

As stated earlier, input documents are processed on a token-by-token basis. Tokens are processed one span (e.g., a sentence or a noun phrase) at a time. Token annotations are specified by the parameter [TokenAnnotation \[9\]](#), while span annotations are specified by the parameter [SpanFeatureStructure \[9\]](#). By default, all tokens within a span are considered, and it is the text associated with each token that is used for lookups. `ConceptMapper` can also be configured to consider tokens differently:

- Case sensitive or insensitive matching. See the parameter [caseMatch \[9\]](#)
- Stop words: ignore token during lookup if it appears in given stop word list. See the parameter [StopWords \[9\]](#)
- Stemming: a stemmer can be specified to be applied to the text of the token. In practice, the stemmer could be a standard stemmer providing the root form of the token, or it could perform other functions, such as abbreviation expansion or spelling variant replacement. See the parameter [Stemmer \[9\]](#)
- Use a token feature instead of the token's text. This is useful for cases where, for example, spelling or case correction results need to be applied instead of the token's original text. See the parameter [TokenTextFeatureName \[10\]](#)

- skip tokens during lookups based on particular feature values, as described below

The ability to skip tokens during lookups based on particular feature values makes it easy to skip, for example, all tokens with particular part of speech tags, or with some previously computed semantic class. For example, given the text below in [Example 2.2, “Sample Input Text” \[5\]](#):

```
Infiltrating mammary carcinoma
```

Example 2.2. Sample Input Text

Assume each word is a token that has a feature `SemanticClass`, and that feature for the token “mammary” contains the value “AnatomicalSite”, while the tokens “Infiltrating” and “carcinoma” do not. It is then possible to configure `ConceptMapper` to indicate that tokens that have a particular feature, in this case `SemanticClass`, equal to one of a set of values, in this case “AnatomicalSite”, should be excluded when performing dictionary lookups (see parameters [ExcludedTokenClasses \[10\]](#) and [ExcludedTokenTypes \[10\]](#)). By doing this, for the purposes of dictionary lookup, the example text would effectively appear to be:

```
Infiltrating carcinoma
```

Example 2.3. Result of Token Skipping

In addition to the set of feature values that indicate their associated token are to be excluded during lookup, there are also configuration parameters that can be used to specify a set of feature values for inclusion (see parameters [IncludedTokenClasses \[10\]](#) and [IncludedTokenTypes \[10\]](#)). The algorithm for selecting annotations to include during lookup is as follows:

```
if there is an includeList but no excludeList
include annotation if feature value in includeList

else if there is an excludeList
exclude annotation if feature value in excludeList

else
include annotation
```

Example 2.4. Token Selection Algorithm

This provides a simple way to restrict the selection of pre-classified tokens, whether that pre-classification is done via previous instances of `ConceptMapper` or some altogether different annotator. See [TokenTextFeatureName \[10\]](#)

2.4. Lookup Strategies

The actual dictionary lookup algorithm is controlled by three parameters. One specifies token-order independent lookup ([OrderIndependentLookup \[10\]](#)). For example, a dictionary entry that contained the variant:

```
<variant base='carcinoma, infiltrating' />
```

would also match against any permutation of its tokens. In this case, assuming that punctuation was ignored, it would match against both “infiltrating carcinoma” and “carcinoma, infiltrating”. Clearly, this particular setting must be used with care to prevent incorrect matches from being found, but for some domains it enables the use of a more compact dictionary, as all permutations of a particular entry do not need to be enumerated.

Another parameter that controls the dictionary lookup algorithm toggles between finding only the longest match vs. finding all possible matches ([FindAllMatches \[11\]](#)). For the text:

```
... carcinoma, infiltrating ...
```

If there was a dictionary entry for “carcinoma” as well as the entry for “carcinoma, infiltrating”, this parameter would control whether only the latter was annotated as a result or both would be annotated. Using the setting that indicates all possible matches should be found is useful when subsequent disambiguation is required.

The final parameter that controls the dictionary lookup algorithm specifies the search strategy ([SearchStrategy \[10\]](#)), of which there are three. The default search strategy only considers contiguous tokens (not including tokens from the stop word list or otherwise skipped tokens, as described above), and then begins the subsequent search after the longest match. The second strategy allows for ignoring non-matching tokens, allowing for disjoint matches, so that a dictionary entry of

```
A C
```

would match against the text

```
A B C
```

This can be used as alternative method for finding “infiltrating carcinoma” over the text “infiltrating mammary carcinoma”, as opposed to the method described above, wherein the token “mammary” had to have been somehow pre-marked with a feature and that feature listed as indicating the token should be skipped. On the other hand, this approach is less precise, potentially finding completely disjoint and unrelated tokens as a dictionary match. As with the default search strategy, the subsequent search begins after the longest match.

The final search strategy is identical to the previous, except that subsequent searches begin one token ahead, instead of after the previous match. This enables overlapped matching. As with the setting that finds all matches instead of the longest match, using this setting is useful when subsequent disambiguation is required.

2.5. Output Options

Output is in the form of new UIMA annotations. As previously discussed, the mapping from dictionary entry attributes to the result annotation features can also be specified. Given the fact that dictionary entries can have multiple variants, and that matches could contain non-contiguous sets of tokens, it can be useful to be able to know exactly what was matched. There are two parameters that can be used to provide this information. One allows the specification of a feature in the output annotation that will be set to the string containing the matched text. The other can be used to indicate a feature to be filled with the list of tokens that were matched. Going back to the

example in figure 2, where the token “mammary” was skipped, the matched string would be set to “Infiltrating carcinoma” and the matched tokens would be set to the list of tokens “Infiltrating” and “carcinoma”.

Another output control AE descriptor parameter can be used to specify a feature of the resultant annotation to be set to contain the span annotation enclosing the matched token. Assuming, for example, that the spans being processed are sentences, this provides a convenient way to link the resultant annotation back to its enclosing sentence.

It is also possible to indicate dictionary attributes to store back into each of the matched tokens. This provides the ability for tokens to be marked with information regarding what it was matched against. Going back to the example in figure 2, one way that the SemanticClass feature of the token “mammary” could have been labeled with the value “AnatomicalSite” was using this technique: a previous invocation of ConceptMapper had “mammary” as a dictionary entry, that entry had the SemanticClass feature with the value “AnatomicalSite”, and SemanticClass was listed as an attribute to write back as a token feature. If, instead of “mammary” the match was against a multi-token entry, then each of the multiple tokens would have that feature set.

Chapter 3. Configuration Parameters

Detailed description of all configuration parameters:

- `TokenizerDescriptorPath`: *[Required]* String
Path to tokenizer Analysis Engine descriptor, which is used to tokenize dictionary entries. This path may be a reference to a file in the file system, or a reference to an item in the classpath.
- `LanguageID`: *[Required]* String
Language ID (ISO 639-2), for use by the tokenizer specified by [TokenizerDescriptorPath](#) [9].
- `TokenAnnotation`: *[Required]* String
Type of feature structure representing tokens in the input CAS.
- `SpanFeatureStructure`: *[Required]* String
Type of feature structure that corresponds to spans of data for processing (e.g. a sentence) in the input CAS.
- `AttributeList`: *[Required]* Array of Strings
List of attribute names for XML dictionary entry record. Must correspond to parallel list [FeatureList](#) [9].
- `FeatureList`: *[Required]* Array of Strings
List of feature names for [ResultingAnnotationName](#) [11]. Must correspond to parallel list [AttributeList](#) [9].
- `caseMatch`: *[Required]* String
Specifies the case folding mode. The following are the allowable values:
 - `ignoreall` - fold everything to lowercase for matching
 - `insensitive` - fold only tokens with initial caps to lowercase
 - `digitfold` - fold all (and only) tokens with a digit
 - `sensitive` - perform no case folding
- `StopWords`: *[Optional]* Array of Strings
A list of words that are always to be ignored in dictionary lookups.
- `Stemmer`: *[Optional]* String
Name of stemmer class to use before matching. *Must* implement the `org.apache.uima.conceptMapper.support.stemmer` interface and have a zero-parameter constructor. If not specified, no stemming will be performed.

-
- `TokenTextFeatureName`: *[Optional]* String
Name of feature of token annotation that contains the token's text. If not specified, the token's covered text will be used.
 - `TokenClassFeatureName`: *[Optional]* String
Name of feature used when doing lookups against [IncludedTokenClasses \[10\]](#) and [ExcludedTokenClasses \[10\]](#). Values contained in this feature are of type String. This parameter is mandatory if either [IncludedTokenClasses \[10\]](#) or [ExcludedTokenClasses \[10\]](#) are specified. See [Example 2.4, "Token Selection Algorithm" \[5\]](#) for a description of how these are used during lookup.
 - `TokenTypeFeatureName`: *[Optional]* String
Name of feature used when doing lookups against [IncludedTokenTypes \[10\]](#) and [ExcludedTokenTypes \[10\]](#). Values contained in this feature are of type Integer. This parameter is mandatory if either [IncludedTokenTypes \[10\]](#) or [ExcludedTokenTypes \[10\]](#) are specified. See [Example 2.4, "Token Selection Algorithm" \[5\]](#) for a description of how these are used during lookup.
 - `IncludedTokenTypes`: *[Optional]* Array of Integers
Type of tokens to include in lookups (if not supplied, then all types are included except those specifically mentioned in [ExcludedTokenTypes \[10\]](#))
 - `ExcludedTokenTypes`: *[Optional]* Array of Integers
Type of tokens to exclude from lookups (if not supplied, then all types are excluded except those specifically mentioned in [IncludedTokenTypes \[10\]](#), unless [IncludedTokenTypes \[10\]](#) is not supplied, in which case none are excluded)
 - `IncludedTokenClasses`: *[Optional]* Array of Strings
Class of tokens to include in lookups (if not supplied, then all classes are included except those specifically mentioned in [ExcludedTokenClasses \[10\]](#))
 - `ExcludedTokenClasses`: *[Optional]* Array of Strings
Class of tokens to exclude from lookups (if not supplied, then all classes are excluded except those specifically mentioned in [IncludedTokenClasses \[10\]](#), unless [IncludedTokenClasses \[10\]](#) is not supplied, in which case none are excluded).
 - `OrderIndependentLookup`: *[Optional]* Boolean
If "True", token (as specified by [TokenAnnotation \[9\]](#)) ordering within span (as specified by [SpanFeatureStructure \[9\]](#)) is ignored during lookup (i.e., "top box" would equal "box top"). Default is False.
 - `SearchStrategy`: *[Optional]* String
Specifies the dictionary lookup strategy. The following are the allowable values:
 - `ContiguousMatch` - longest match of contiguous tokens (as specified by [TokenAnnotation \[9\]](#)) within enclosing span (as specified by [SpanFeatureStructure \[9\]](#)), taking into account included/excluded items (see [IncludedTokenTypes \[10\]](#), [ExcludedTokenTypes \[10\]](#),

[IncludedTokenClasses \[10\]](#) and [ExcludedTokenClasses \[10\]](#)). DEFAULT strategy

- `SkipAnyMatch` - longest match of not-necessarily contiguous tokens (as specified by [TokenAnnotation \[9\]](#)) within enclosing span (as specified by [SpanFeatureStructure \[9\]](#)), taking into account included/excluded items (see [IncludedTokenTypes \[10\]](#), [ExcludedTokenTypes \[10\]](#), [IncludedTokenClasses \[10\]](#) and [ExcludedTokenClasses \[10\]](#)). Subsequent lookups begin in span after complete match. *Implies* order-independent lookup (see [OrderIndependentLookup \[10\]](#)).
- `SkipAnyMatchAllowOverlap` - longest match of not-necessarily contiguous tokens (as specified by [TokenAnnotation \[9\]](#)) within enclosing span, (as specified by [SpanFeatureStructure \[9\]](#)) taking into account included/excluded items (see [IncludedTokenTypes \[10\]](#), [ExcludedTokenTypes \[10\]](#), [IncludedTokenClasses \[10\]](#) and [ExcludedTokenClasses \[10\]](#)). Subsequent lookups begin in span after next token. *Implies* order-independent lookup (see [OrderIndependentLookup \[10\]](#)).
- `FindAllMatches`: *[Optional]* Boolean
If True, all dictionary matches are found within the span specified by [SpanFeatureStructure \[9\]](#), otherwise only the longest matches are found.
- `ResultingAnnotationName`: *[Optional]* String
Name of the annotation type created by this TAE.
- `ResultingEnclosingSpanName`: *[Optional]* String
Name of the feature in the [ResultingAnnotationName \[11\]](#) that will be set to point to the span annotation that encloses it (i.e. its sentence)
- `ResultingAnnotationMatchedTextFeature`: *[Optional]* String
Name of the feature in the [ResultingAnnotationName \[11\]](#) that will be set to the string that was matched in the dictionary. This could be different than the annotation's covered text if there were any skipped tokens in the match.
- `MatchedTokensFeatureName`: *[Optional]* String
Name of the FSArray feature in the [ResultingAnnotationName \[11\]](#) that will set to the set of tokens matched.
- `TokenClassWriteBackFeatureNames`: *[Optional]* Array of Strings
Names of features in the [ResultingAnnotationName \[11\]](#) that should be written back to a token from the matching dictionary entry, such as a POS tag.
- `PrintDictionary`: *[Optional]* Boolean
If True, print dictionary after loading. Default is False.
- `DictionaryFile`: *[Dictionary Resource]* Boolean
Dictionary file resource specification. Specify class name for dictionary loader, then bind to name of file containing dictionary contents to be loaded.

