ANNIS2 User Guide – Version 2.1.8

(For the latest documentation see also: http://korpling.german.hu-berlin.de/trac)

Contents

1 Introduction

ANNIS2 is an open source, browser-based search and visualization architecture for multi-layer corpora. It can be used to search for complex graph structures of annotated nodes and edges forming a variety of linguistic structures, such as constituent or dependency syntax trees, coreference and parallel alignment edges, span annotations and associated multi-modal data (audio/video). This guide provides an overview of the current ANNIS2 system, first steps for installing either a local instance or an ANNIS server with a demo corpus, as well as tutorials for converting data for ANNIS and running queries with AQL (ANNIS Query Language).

2 New Features in Version 2.1.8

Performance:

- Massive acceleration of query times through a completely restructured DB scheme
- Faster corpus import times by using more efficient indexes

- Import speed is no longer dependent on current DB size/previous corpora thanks to partitioning

Visualization:

- Three new dependency tree visualizations:
  - ordered arch dependency visualizer (courtesy of Kim Gerdes, ILPGA, Paris)
  - unordered and ordered tree dependency visualizers (developed in conjunction with Dag Haug / PROIEL project, Oslo)
- Debug graph visualization (a graph of all annotations, not very readable but useful for debugging)
- Improved discourse/coreference visualizer now handles nested discourse referents and displays edge annotations
- Improved grid visualizer handles overlapping spans with same annotation name correctly
- Differential match coloring (each search node is highlighted in a different color)

Search and export:

- Addition of generations for pointing relations (#1 ->anaphor 3,4 #2 finds chains of length 3 to 4)
- New exporters for tokens with their annotations and flattened grids in plain text
- Negation in metadata works correctly (bugfix)
- Backend timeout works correctly (bugfix)
- Simplification of AQL-SQL generation (mainly relevant for developpers)

(For change logs of previous version see their respective distributions or user guides)

3 Installing ANNIS2

3.1 Installing a Local Version (ANNIS Kickstarter)

Local users who do not wish to make their corpora available online can install ANNIS Kickstarter. To install Kickstarter follow these steps:

1. Download and install PostgreSQL 8.4 for your operating system from http://www.postgresql.org/download/ and **make a note of the administrator password** you set during the installation. After installation, Postgres may automatically launch the Postgres Stack Builder to download additional components – you can safely skip this step and cancel the Stack Builder if you wish. You may need to restart your OS if the Postgres installer tells you to.
2. Download and unzip Annis-Kickstarter-2.1.8.zip from the ANNIS website.
3. Start AnnisKickstarter.bat if you're using Windows or run the bash script AnnisKickstarter.sh otherwise (this may take a few seconds the first time you run Kickstarter). At this point your Firewall may try to block Kickstarter and offer you to unblock it – do so and Kickstarter should start up.

Note: for most users it is a good idea to give Java more memory (if this is not already the default). You can do this by editing the script AnnisKickstarter and typing the following after the call to start java (before -splash:splashscreen.gif):

-Xss1024k -Xmx1024m

(To accelerate searches it is also possible to give the Postgres database more memory, see the link in the next section below).

4. Once the program has started, if this is the first time you run Kickstarter, press "Init Database" and supply your PostGres administrator password from step 1.
5. Download and unzip the pcc2 demo corpus from the ANNIS website.
6. Press "Import Corpus" and navigate to the directory containing the directory pcc2_relAnnis/. Select this directory (but do not go into it) and press OK.
7. Once import is complete, press "Launch Annis frontend" and login with the username and password "test" to test the corpus (try selecting the pcc2 corpus, typing pos="NN" in the AnnisQL box and clicking "Show Result". See the section "Running Queries in ANNIS2" in this guide for some more example queries, or press the Tutorial button at the top left of the interface).

3.2 Building and Installing an ANNIS Server

The ANNIS server version can be installed on UNIX based server, or else under Windows using Cygwin, the freely available UNIX emulator. To install the ANNIS server:

1. Install a PostgreSQL server for your operating system from http://www.postgresql.org/download/
2. Install a web server such as Tomcat or Jetty
3. Make sure you have JDK 6 and Maven 2 (or install them if you don't)
4. If you're using Cygwin and Windows you will also need to install the "patch" program via the Cygwin package manager
5. Download and unzip Annis-2.1.8.zip, then run the following commands (replacing the appropriate directories):

```
cd <unzipped source>/Annis-Service

mvn -DskipTests=true install

mvn -DskipTests=true assembly:assembly

tar xzvf target/annis-service-<version>-distribution.tar.gz -C <installation directory>
```

6. Next initialize your ANNIS database (only the first time you use the system):

7. Set the environment variables (each time when starting up)

```
export ANNIS_HOME=<installation directory>
```

```
export PATH=$PATH:$ANNIS_HOME/bin
```

8. Now you can import some corpora:

```
annis-admin.sh import path/to/corpus1 path/to/corpus2 ...
```

Important: The above import-command calls other PostgreSQL database commands. If you abort the import script with Ctrl+C, these SQL processes will not be automatically terminated; instead they might keep hanging and prevent access to the database. The same might happen if you close your shell before the import script terminates, so you will want to prefix it with the "nohup"-command.

9. Now you can start the ANNIS service:

```
annis-service.sh start
```

10. To get the Annis front-end running, first compile it:

```
cd <unzipped source>

mvn -DskipTests=true install
```

If no error occurs the war-file will be available under <unzipped source>/Annis-web/target/Annis-web.war.

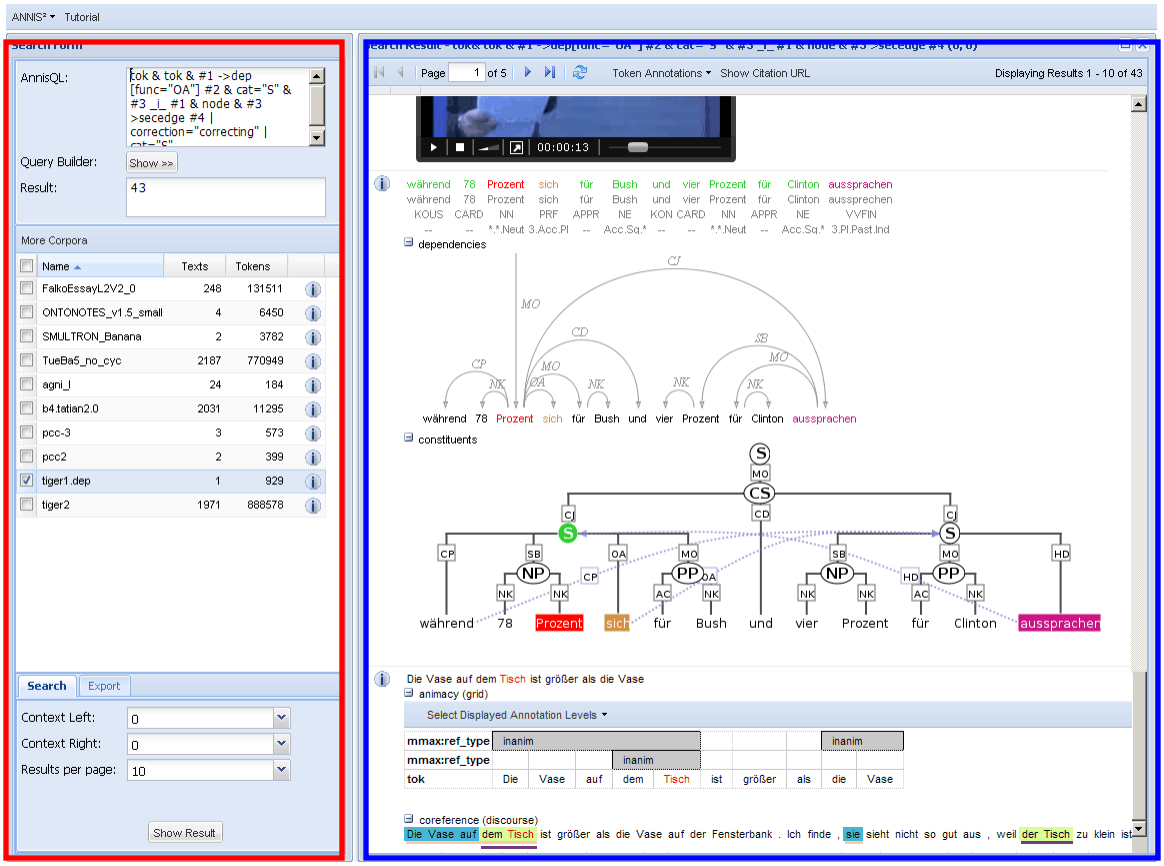
11. And configure your web server as described here: http://korpling.german.hu-berlin.de/trac/annis/wiki/Documentation/Web/Tomcat

The latest instructions for compiling and installing the ANNIS Server can also be found at: http://korpling.german.hu-berlin.de/trac/annis/wiki/Documentation

We also **strongly recommend** reconfiguring the Postgres server's default settings as described here:
http://korpling.german.hu-berlin.de/trac/annis/wiki/Documentation/Service/PostgreSQL

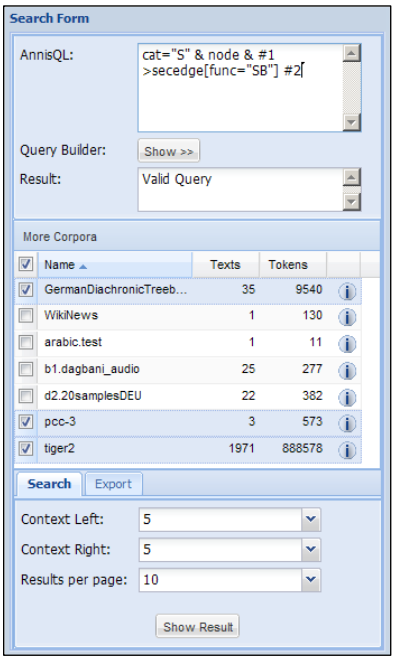
4 Running Queries in ANNIS2

4.1 The ANNIS2 Interface



The ANNIS2 interface is comprised of several windows, the most important of which are the search form (in the red box above) and the results window (in the blue box above).

The Search Form



The Search Form on the left of the interface window is available immediately after login. In the middle, the list of currently available corpora is shown. Using the checkboxes on the left of each corpus, it is possible to select which corpora should be searched in (hold down 'shift' to select multiple corpora simultaneously). If you cannot see a corpus that should be available to you, or else if the corpora list is too cluttered, you may click on "more corpora" to open the corpora window. You may then drag and drop the desired or unwanted corpora between the list and the window.

The "AnnisQL" field at the top of the form is used for inputting queries manually (see the tutorials on the ANNIS Query Language). As soon as a one or several

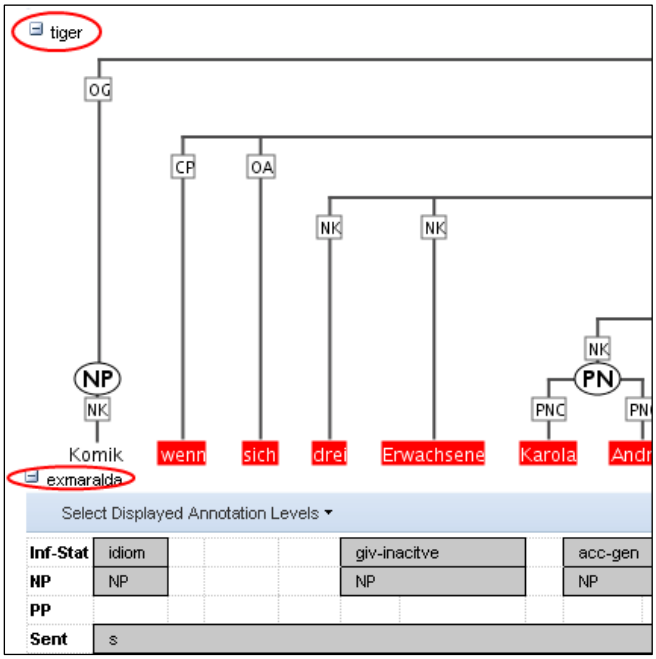
corpora are selected and a query is entered or modified, the query will be validated automatically and possible errors in the query syntax will be commented on in the "Result" box below. When modifying a query, a delay of two seconds ia activated before the query is re-sent to the server for validation.

Once a valid query has been entered, pressing the "Show Result" button will retrieve the number of matching positions in the selected corpora in the Result box and open the Result Window to display the first set of matches. The context surrounding the matching expressions in the result list ist determined by the "context left" and "context right" options at the bottom of the search form, and can be set to up to 10 tokens on each side, though some corpora allow longer spans, such as entire texts, to be viewed using special discourse visualizations.

The Result Window

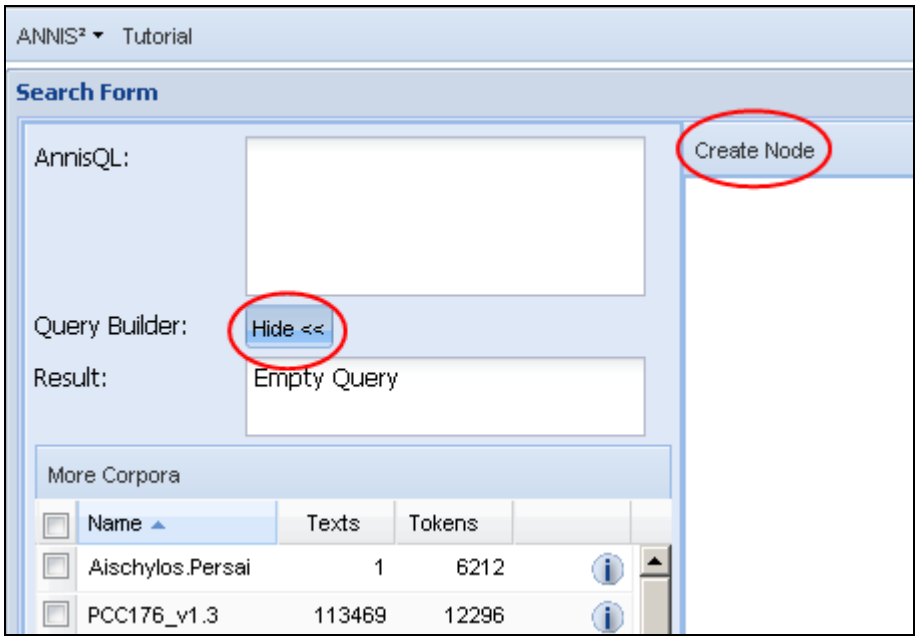
The result window shows search results in pages of 10 hits each by default (this can be changed in the Search Form). The toolbar at the top of the window allows you to navigate between these pages. The "Token Annotations" button on the toolbar allows you to toggle the token based annotations, such as lemmas and parts-of-speech, on or off for you convenience. The "Citation URL" button provides a hyperlink which you can e-mail or cite, allowing others to reproduce your query.

The result list itself initially shows a KWIC (key word in context) concordance of matching positions in the selected corpora, with the matching region marked red and the context in black on either side. Token annotations are displayed in gray under each token, and hovering over them with the mouse will show the annotation name and namespace. More complex annotation levels can be expanded, if available, by clicking on the plus icon next to the level's name, e.g. tiger and exmaralda for the annotations in the tree and grid views in the picture to the right (circled in red).

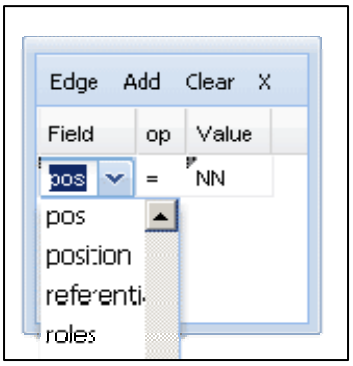


4.2 Using the ANNIS2 Query Builder

To open the graphical query builder, click on the Query Builder: Show >> button on the Search Form (then clicking Query Builder: hide << will close the Query Builder). On the left-hand side of the toolbar at the top of the query builder canvans, you will see the Create Node button. Use this button to define nodes to be searched for (tokens, non-terminal nodes or annotations). Creating nodes and modifying them on the canvas will immediately update the AnnisQL field in the Search Form with your query, though updating the query on the Search Form will not create a new graph in the Query Builder.

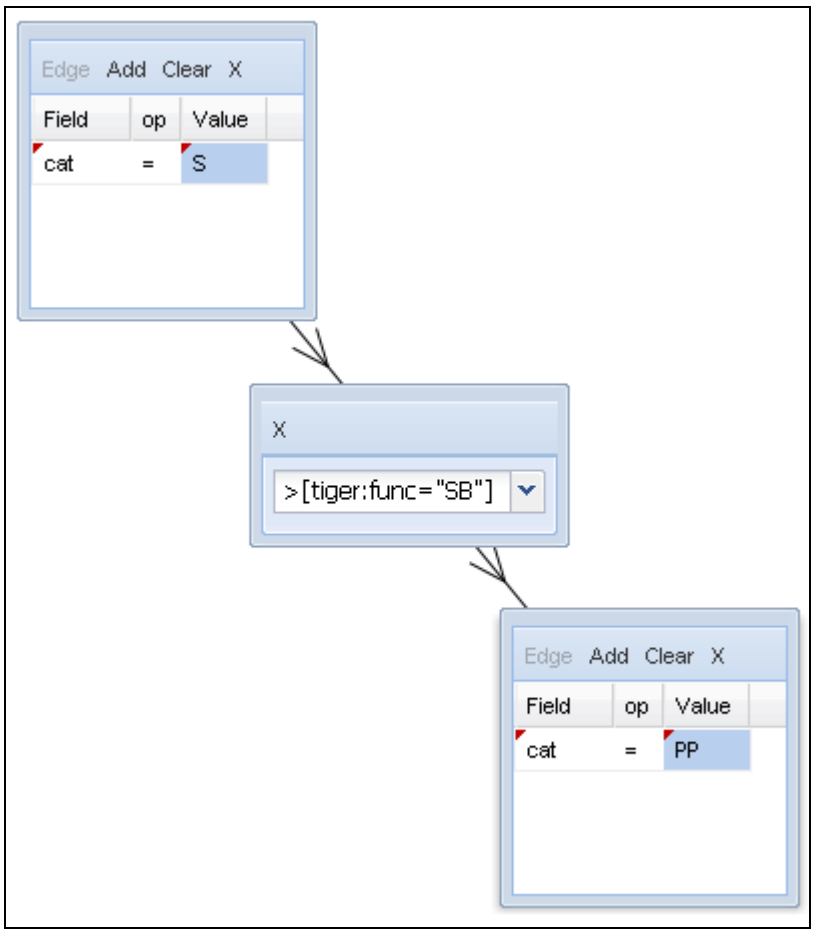


In each node you create you may click on "Add" to specify an annotation value. The annotation name can be typed in or selected from a drop down list. The "Op[erator]" field in the middle allows you to choose between an exact match (the '=' symbol) or wildcard search using Regular Expressions (the '~' symbol). The annotation value is given on the right, and should NOT be surrounded by quotations (see the example below). It is also possible to specify multiple annotations applying to the same position by clicking on "Add" multiple times. Clicking on "Clear" will delete the values in the node. To search for word forms, simply leave the field name on the left empty and type directly on the right under "Value". A node with no data entered will match any node, that is an underspecified token or non-terminal node or annotation.



To specify the relationship between nodes, first click on the "Edge" button at the top left of one node, and then click the "Dock" button which becomes available on the other nodes. An edge will connect the nodes with an extra box from which operators may be selected (see below). For operators allowing additional labels (e.g. the dominance operator >

allows edge labels to be specified), you may type directly into the edge's operator box, as in the example with a "func" label in the image below. Note that the node clicked on first (where the "Edge" button was clicked) will be the first node in the resulting quey, i.e. if this is the first node it will dominate the second node (#1 > #2) and not the other way around, as also represented by the arrows along the edge.



4.3 Searching for Word Forms

To search for word forms in ANNIS2, simply select a corpus (in this example the small PCC2 demo corpus) and enter a search string between double quotation marks, e.g.:

```
"statisch"
```

Note that the search is case sensitive, so it will not find cases of capitalized 'Statisch', for example at the beginning of a sentence. In order to find both options, you can either look for one form OR the other using the pipe sign (|):

```
"statisch" | "Statisch"
```

or else you can use regular expressions, which must be surrounded by slashes (/) instead of quotation marks:

```
/[Ss]tatisch/
```

To look for a sequence of multiple word forms, enter your search terms separated by & and then specify that the relation between the elements is one of precedence, as signified by the period (.) operator:

```
"so" & "statisch" & #1 . #2
```

The expression #1 . #2 signifies that the first element ("so") precedes the second element ("statisch"). For indirect precedence (where other tokens may stand between the search terms), use the .* operator:

```
/[Ss]o/ & "statisch" & "wie" & #1 . #2 & #2 .* #3
```

The above query finds sequences beginning with either "So" or "so", followed directly by "statisch", which must be followed either directly or indirectly (.*) by "wie". A range of allowed distances can also be specified numerically as follows:

```
/[Ss]tatisch/ & "wie" & #1 .1,5 #2
```

Meaning the two words may appear at a distance of 1 to 5 tokens. The operator .* allows a distance of up to 50 tokens by default, so searching with .1,50 is the same as using .* instead. Greater distances (e.g. .1,100 for 'within 100 tokens') should always be specified explicitly.

Finally, we can add metadata restrictions to the query, which filter out documents not matching our definitions. Metadata attributes must be preceded by the prefix meta:: and may not be bound (i.e. they are not referred to as #1 etc. and the numbering of other elements ignores their existence):

```
/[Ss]tatisch/ & "wie" & #1 .1,5 #2 & meta::Genre="Sport"
```

To view metadata for a search result or for a corpus, press the "i" icon next to it in the result window or in the search form respectively.

4.4 Searching for Annotations

Annotations may be searched for using an annotation name and value. The names of the annotations vary from corpus to corpus, though many corpora contain part-of-speech and lemma annotations with the names pos and lemma respectively (annotation names are case sensitive). For example, to search for all forms of the German verb sein 'to be' in a corpus with lemma annotation such as PCC2, simply select the PCC2 corpus and enter:

```
lemma="sein"
```

Negative searches are also possible using != instead of =. For negated tokens (word forms) use the reserved attribute tok. For example:

```
lemma!="sein"
```

or:

```
tok!="ist"
```

Metadata can also be negated similarly:

```
lemma="sein" & meta::Genre!="Sport"
```

To only find finite forms of this verb in PCC2, use the part-of-speech (pos) annotation concurrently, and specify that both the lemma and pos should apply to the same element:

```
lemma="sein" & pos="VAFIN" & #1 _=_ #2
```

The expression #1 _=_ #2 uses the span identity operator to specify that the first annotation and the second annotation apply to exactly the same position in the corpus. Annotations can also apply to longer spans than a single token: for example, in PCC2, the annotation Inf-Stat signifies the information structure status of a discourse referent. This annotation can also apply to phrases longer than one token. The following query finds spans containing new discourse referents, not previously mentioned in the text:

```
exmaralda:Inf-Stat="new"
```

If the corpus contains no more than one annotation type named Inf-Stat, the optional namespace (in this case exmaralda:) may be dropped; if there are multiple annotations with the same name but different namespaces, dropping the namespace will find all of those annotations. In order to view the span of tokens to which this annotation applies, enter the and click on "Show Result", then open the exmaralda annotation level to view the grid containing the span. Further operators can test the relationships between potentially overlapping annotations in spans. For example, the operator _i_ examines whether one annotation fully contains the span of another annotation (the i stands for 'includes'):

```
Topic="ab" & Inf-Stat="new" & #1 _i_ #2
```

This query finds aboutness topics (Topic="ab") containing information structurally new discourse referents.

4.5 Searching for Trees

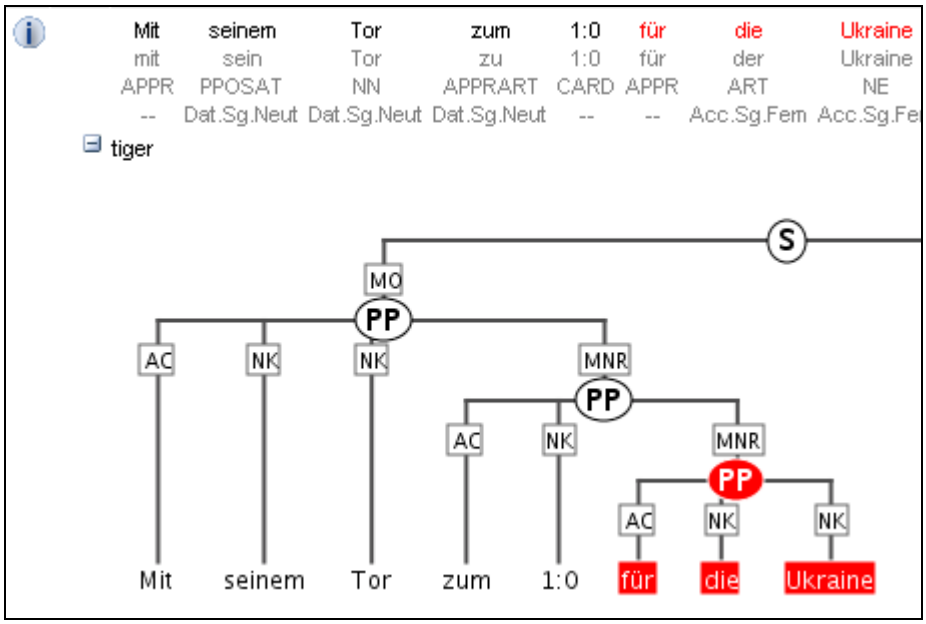
In corpora containing hierarchical structures, annotations such as syntax trees can be searched for by defining terminal or none-terminal node annotations and their values. A simple search for prepostional phrases in the small PCC2 demo corpus looks like this:

```
tiger:cat="PP"
```

If the corpus contains no more than one annotation called cat, the optional namespace, in this case tiger:, may be dropped. This finds all PP nodes in the corpus. To find all PP nodes directly dominating a proper name, a second element can be specified with the appropriate part-of-speech (pos) value:

```
cat="PP" & pos="NE" & #1 > #2
```

The operator > signifies direct dominance, which must hold between the first and the second element. Once the Result Window is shown you may open the "tiger" annotation level to see the corresponding tree.



Note that since the context is set to a number of tokens left and right of the search term, the tree for the whole sentence may not be retrieved. To do this, you may want to specifically search for the sentence dominating the PP. To do so, specify the sentence in another element and use the indirect dominance (>*) operator:

```
cat="S" & cat="PP" & pos="NE" & #1 >* #2 & #2 > #3
```

If the annotations in the corpus support it, you may also look for edge labels. Using the following query will find all adjunct modifiers of a VP, dominated by the VP node through an edge labeled MO. Since we do not know anything about the modifying node, whether it is a non-terminal node or a token, we simply use the node element as a place holder. This element can match any node or annotation in the graph:

```
cat="VP" & node & #1 >[tiger:func="MO"] #2
```

It is also possible to negate the label of the dominance edge as in the following query:

```
cat="VP" & node & #1 >[tiger:func!="MO"] #2
```

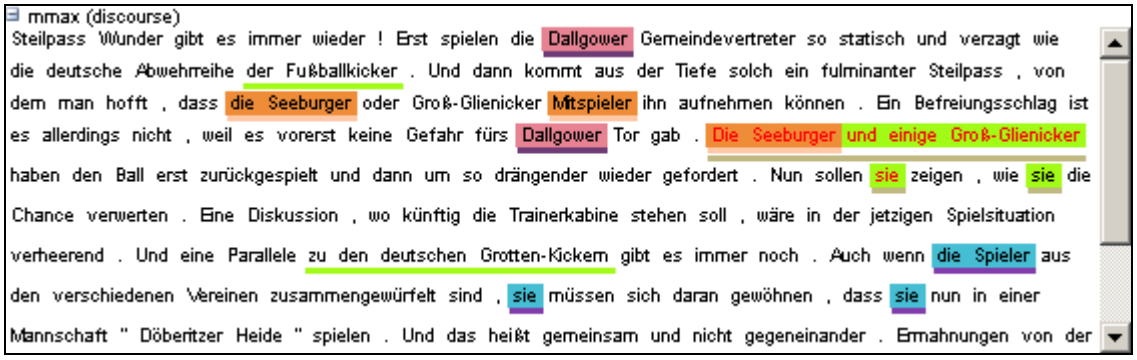
which finds all VPs dominating a node with a label other than MO.

4.6 Searching for Pointing Relations – Coreference and Dependencies

Pointing relations are used to express an arbitrary directed relationship between two elements (terminals or non-terminals) without implying dominance or coverage inheritance. For instance, in the pcc2 demo corpus, elements in the mmax: namespace may point to each other to express coreference or anaphoric relations. The following query searches for two np_form annotations, which specify for example whether a nominal phrase is pronominal, definite or indefinite.

```
mmax:np_form="pper" &
mmax:np_form="defnp" &
#1 ->anaphor_antecedent #2
```

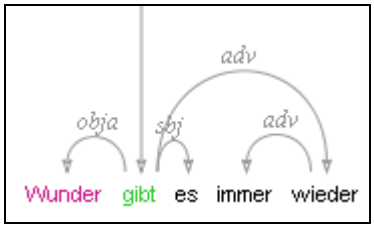
Using the pointing relation operator -> with the type anaphor_antecedent, the first np_form, which should be a personal pronoun (pper), is said to be the anaphor to its antecedent, the second np_form, which is definite (defnp). To see a visualization of the coreference relations, open the mmax annotation level in the example corpus. In the image below, one of the matches for the above query is highlighted in red (*die Seeburger und einige Groß-Glienicker ... sie* 'the Seeburgers and some Groß-Glienickers... they'). Other discourse referents in the text (marked with an underline) may be clicked on, causing coreferential chains containing them to be highlighted as well. Note that discourse referents may overlap, leading to multiple underlines: *Die Seeburger* 'the Seeburgers' is a shorter discourse referent overlapping with the larger one ('the Seeburgers and some Groß-Glienickers'), and each referent has its own underline. Annotations of the coreference edges of each relation can be viewed by hovering of the appropriate underline.



Another way to use pointing relations is found in syntactic dependency trees. The queries in this case can use both pointing relation types and annotation, as in the following query:

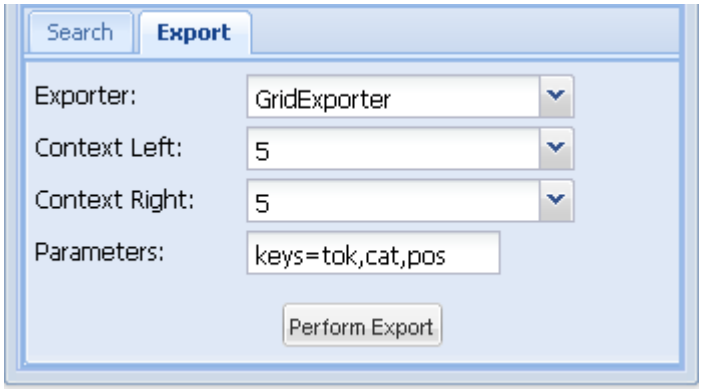
```
pos="VVFIN" & tok & #1 ->dep[func="objа"] #2
```

This query searches for a finite verb (with the part-of-speech VVFIN) and a token, with a pointing relation of the type ‘dep’ (for dependency) between the two, annotated with ‘func="obja"’ (the function Object, Accusative). The result can be viewed with the dependency arch visualizer, which shows the verb *gibt* ‘gives’ and its object *Wunder* ‘miracles’.



4.7 Exporting Search Results

By going to the Export tab at the bottom of the search form on the left, you can select one of several exporters:



The **SimpleTextExporter** simply gives the text for all tokens in each search result, including context, in a one-row-per-hit format. The tokens covered by the match area are marked with square brackets and the results are numbered, as in the following example:

1. Tor zum 1:0 für die [Ukraine] stürzte der 1,62 Meter große
2. der 1,62 Meter große Gennadi [Subow] die deutsche Nationalelf vorübergehend in
3. und Reputation kämpfenden Mannschaft von [Rudi] Völler der Weg zur Weltmeisterschaft
4. Reputation kämpfenden Mannschaft von Rudi [Völler] der Weg zur Weltmeisterschaft endgültig
5. die deutschen Nationalkicker einen " [Rudi] Riese " auf der Bank

The **TextExporter** adds all annotations of each token separated by slashes (e.g. `dogs/NN/dog` for the token dogs annotated with a part-of-speech NN and a lemma dog).

The **GridExporter** adds all annotations available for the span of retrieved tokens, with each annotation layer in a separate line. Annotations are separated by spaces and the hierarchical order of annotations is lost, though the span of tokens covered by each

annotation may optionally be given in square brackets (to turn this off use the optional parameter `numbers=false`). The user can specify annotation layers to be exported in the additional 'Parameters' box, using the setting 'keys=' and annotation names separated by comas. If nothing is specified in the parameters box, all available annotations will be exported. Multiple options are separated by a semicolon, e.g. `keys=tok,pos,cat;numbers=false`. An example output with token numbers looks as follows.

1.	tok	ein Dialog zwischen den Generationen angestoßen .
	cat	NP[1-5] S[1-6] VP[1-6] PP[3-5]
	pos	ART[1-1] NN[2-2] APPR[3-3] ART[4-4] NN[5-5] VVPP[6-6] $.[7-7]

Meaning that the annotation cat="NP" applied to tokens 1-5 in the search result, and so on. Note that when specifying annotation layers, if the reserved name **'tok'** is not specified, the tokens themselves will not be exported (annotations only).

The **WekaExporter** outputs the format used by the WEKA machine learning tool (http://www.cs.waikato.ac.nz/ml/weka/). Only the attributes of the search elements (#1, #2 etc. in AQL) are outputted, and are separated by commas. The order and name of the attributes is declared in the beginning of the export text, as in this example:

```
@relation name

@attribute #1_id string
@attribute #1_token string
@attribute #1_tiger:cat string
@attribute #2_id string
@attribute #2_token string
@attribute #2_tiger:lemma string
@attribute #2_tiger:morph string
@attribute #2_tiger:pos string

@data

'288662','NULL','NP','288392','ganze','ganz','Pos.Acc.Sg.Fem','ADJA'
'289175','NULL','NP','288712','geladenen','geladen','Pos.Nom.Pl.*','ADJA'
'289660','NULL','NP','289409','Döberitzer','Döberitzer','Pos.*.*.*','ADJA'
'288672','NULL','NP','288302','deutschen','deutsch','Pos.Nom.Pl.Masc','ADJA'
'289614','NULL','NP','289291','deutsche','deutsch','Pos.Nom.Sg.Fem','ADJA'
'289625','NULL','NP','289245','fulminanter','fulminant','Pos.Nom.Sg.Masc','ADJA'
'288607','NULL','NP','288242','einstige','einstig','Pos.Nom.Sg.Fem','ADJA'
'288620','NULL','NP','288334','ähnliche','ähnlich','Pos.Acc.Pl.Neut','ADJA'
'289220','NULL','NP','288883','große','groß','Pos.Nom.Sg.Fem','ADJA'
'288610','NULL','NP','288313','deutsche','deutsch','Pos.Acc.Sg.Fem','ADJA'
'289174','NULL','NP','288809','böse','böse','Pos.Nom.Sg.Fem','ADJA'
'289611','NULL','NP','289241','Dallgower','Dallgower','Pos.*.*.*','ADJA'
'288624','NULL','NP','288330','ukrainische','ukrainisch','Pos.Nom.Sg.Masc','ADJA'
```

The export shows the properties of an NP node dominating a token with the part-of-speech ADJA. Since the token also has other attributes, such as the lemma, the token text and morphology, these are also retrieved. Note that exporting may be slow in both exporters if the result set is very large.

4.8 Complete List of Operators

The ANNIS Query Language (AQL) currently includes the following operators:

Operator	Description	Illustration	Notes
.	direct precedence	**A B**	For non-terminal nodes, precedence is determined by the right most and left most terminal children
.*	indirect precedence	**A** x y z **B**	For specific sizes of precedence spans, `.n,m` can be used, e.g. `.3,4` - between 3 and 4 token distance
>	direct dominance	**A** | **B**	A specific edge type may be specifed, e.g.: `>secedge` to find secondary edges. Edges labels are specified in brackets, e.g. `>[func="OA"]` for an edge with the function 'object, accusative'
>*	indirect dominance	**A** | ... | **B**	For specific distance of dominance, `>n,m` can be used, e.g. `>3,4` - dominates with 3 to 4 edges distance
_=_	identical coverage	**A** **B**	Applies when two annotation cover the exact same span of tokens
_i_	inclusion	**AAA** **B**	Applies when one annotation covers a span identical to or larger than another
_o_	overlap	**AAA** **BBB**	For overlap only on the left or right side, use `_ol_` and `_or_` respectively
_l_	left aligned	**AAA** **BB**	Both elements span an area beginning with the same token
_r_	right aligned	**AA** **BBB**	Both elements span an area ending with the same token
->LABEL	labeled pointing relation	LABEL **A** **B**	A labeled, directed relationship between two elements. Annotations can be specified with `->LABEL[annotation="VALUE"]`
->LABEL *	indirect pointing relation	LABEL LABEL **A** ... **B**	An indirect labeled relationship between two elements. The length of the chain may be specified with `->LABEL n,m` for relation chains of length n to m

>@l	left-most child	**A** / | \ **B** x y	
>@r	right-most child	**A** / | \ x y **B**	
$	Common parent node	x / \ **A B**	
$*	Common ancestor node	x | ... / \ **A B**	
#x:arity=n	Arity	x / | \ **1 ... n**	Specifies the amount of directly dominated children that the searched node has
#x:length=n	Length	x ... / \ **1 ... n**	Specifies the length of the span of tokens covered by the node
#x:root	Root	**x** ... / \ 1 ... n	node x is the root of a subgraph (i.e. it is not dominated by any node)

5 Configuring Visualizations with the Resolver Table

5.1 Triggering Visualizations

By default, ANNIS2 displays all search results in the Key Word in Context (KWIC) view in the search result window. Further visualizations, such as syntax trees or grid views, are displayed by default based on the following namespaces:

Nodes with the namespace tiger:	tree visualizer
Nodes with the namespace exmaralda:	grid visualizer
Edges with the namespace mmax:	discourse view
Nodes with the namespace external:	multimedia player

In these cases the namespaces are usually taken from the source format in which the corpus was generated, and carried over into relAnnis during the conversion. It is also possible to use other namespaces, most easily when working with PAULA XML. In PAULA XML, the namespace is determined by the string prefix before the first period in the file name / paula_id of each annotation layer. In order to manually determine the visualizer and the display name for each namespace in each corpus, the resolver table in the database must be edited. To do so, open PGAdmin (or if you did not install PGAdmin with ANNIS then via PSQL), and access the table *resolver_vis_map* (it can be found in PGAdmin under *PostgreSQL 8.4* > *Databases* > *anniskickstart* > *Schemas* > *public* > *Tables* (for ANNIS servers replace "anniskickstart" with "annis_db"). You may need to give your PostgreSQL password to gain access. Right click on the table and select *View Data* > *View All Rows*. The table should look like this:

Edit Data - PostgreSQL 8.4 (localhost:5432) - anniskickstart - resolver_vis_map

File Edit View Tools Help

No limit

	id [PK] serial	corpus character v	version character var	namespace character var	element character var	vis_type character var	display_name character var	order numeric	mappings character var
1	1			tiger	node	tree	tree	101	
2	2			exmaralda	node	grid	exmaralda	102	
3	3			mmax	node	grid	mmax	103	
4	4			mmax	edge	discourse	coref	104	
5	5			urml	node	old_grid	urml	105	
6	6			external		file	external file	106	
7	7					paula	paula	107	
8	8					paula_text	paula text	108	
9	10	b3.parses.1		bitpar	node	tree	bitpar	1	
10	11	b3.parses.1		lingenio	node	tree	lingenio	2	
11	12	parallel_tree_		tiger-de	node	tree	Syntax (Germar	1	
12	13	parallel_tree_		tiger-en	node	tree	Syntax (English)	2	
13	14	b3.parses.1				discourse	Whole Text	4	
14	15	SMULTRON_E		german	node	tree	Syntax (Germar	1	
15	16	SMULTRON_E		english	node	tree	Syntax (English)	2	
*									

Resolver table (resolver_vis_map)

The columns in the table can be filled out as follows:

- *corpus* determines the corpora for which the instruction is valid (null values apply to all corpora)

- *namespace* specifies relevant namespace which triggers the visualization
- *element* determines if a node or an edge should carry the relevant annotation for triggering the visualization
- *vis_type* determines the visualizer module used and is one of:
  - *tree* (constituent syntax tree)
  - *grid* (annotation grid, with annotations spanning multiple tokens)
  - *old_grid* (deprecated version of *grid*)
  - *discourse* (a view of the entire text of a document, possibly with interactive coreference links)
  - *arch_dependency* (dependency tree with labeled arches between tokens; requires SVG enabled browser, see 5.2)
  - *ordered_dependency* (arrow based dependency visualization for corpora with dependencies between non terminal nodes; requires GraphViz, see 5.2)
  - *hierarchical_dependency* (unordered vertical tree of dependent tokens; requires GraphViz)
  - *graph* (a debug view of the annotation graph; requires GraphViz, see 5.2)
  - *file* (a linked multimedia file)

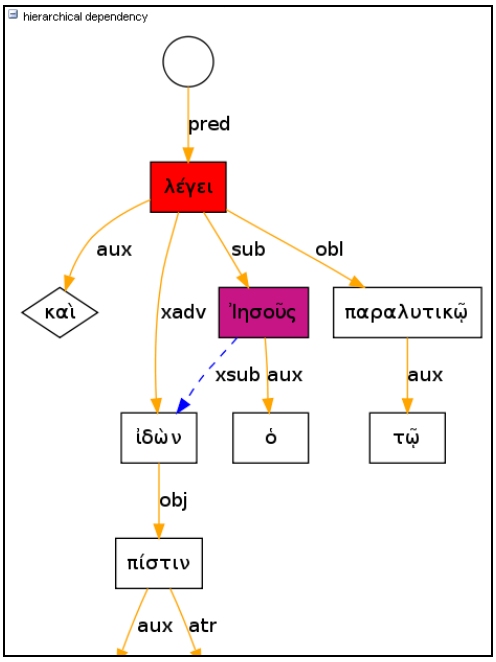
  The additional system internal debug views *paula* and *paula_text* deliver an XML representation of hits and entire texts respectively.
- *display_name* determines the heading that is shown for each visualizer in the interface
- *order* determines the order in which visualizers are rendered in the interface (low to high)
- *mappings* provides additional parameters for some visualizations:
  - *tree* – the annotation names to be displayed in non terminal nodes can be set e.g. using *node_key:cat* for an annotation called *cat* (the default), and similarly the edge labels using *edge_key:func* for an edge label called *func* (the default). Instructions are separated using semicolons.
  - *graph* – use *ns_all:true* to visualize the entire annotation graph. Specifying e.g. *node_ns:tiger* or *edge_ns:tiger* instead causes only nodes and edges of the namespace tiger to be visualized (i.e. only a subgraph of all annotations)
- the field *version* is reserved for future development.

5.2 Visualizations with Software Requirements

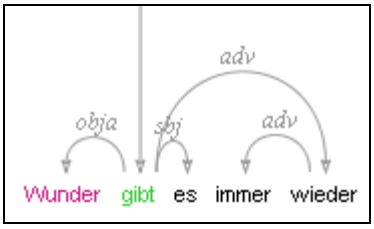
Some ANNIS visualizers require additional software, depending on whether or not they render graphics as an image directly in Java or not. At present, three visualizations require an installation of the freely available software **GraphViz** (http://www.graphviz.org/): *ordered_dependency*, *hierarchical_dependency* and the general *graph* visualization. To use these, install GraphViz on the server (or your local machine for Kickstarter) and make sure it is available in your system path (check this by calling e.g. the program *dot* on the command line).

Another type of restriction is that some visualizers may use **SVG** (scalable vector graphics) instead of images, which mean the user's browser must be SVG capable (e.g.

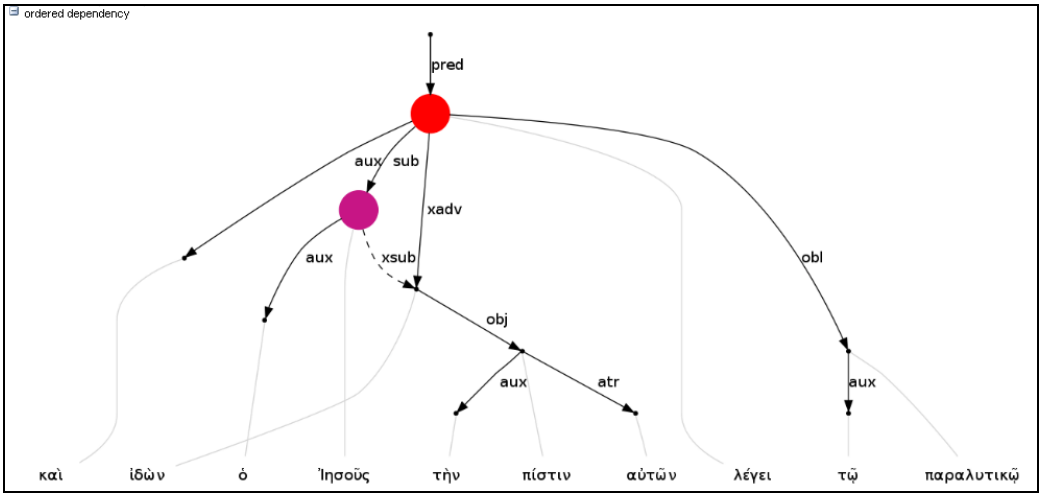
FireFox) or a plugin must be used (e.g. for Internet Explorer 8 or below). This is currently the case for the *arch_dependency* visualizer.



Hierarchical dependency visualizer



Arch dependency visualizer



Ordered tree dependency visualizer

6 Converting Corpora for ANNIS using Pepper 1.0

ANNIS2 uses a relational database format called relANNIS. The Pepper converter framework allows users to convert data from PAULA XML, EXMARaLDA XML, Tiger XML and TreeTagger directly into relAnnis (the Tiger XML conversion is limited to corpora without secondary edges at the moment). Further formats (including Tiger XML with secondary edges) can be converted first into PAULA XML and then into relANNIS using the converters found on the ANNIS downloads page.

6.1 Installing Pepper

Unzip the file Pepper_1.0.0.zip. Pepper is now ready to run. If this does not work correctly, you can compile the sources by running an ANT script (for which you will need to install ANT). With ANT installed, change the directory to your PEPPER_HOME and run ant -f build.xml.

6.2 Running Pepper

To run Pepper you have to create a workflow containing the steps to be carried out during the conversion process. The workflow should be described in an xml-file (called Pepper-workflow or Pepper-params). To run the program you must assign the workflow-file by using the flag -p in program call. The following example shows the usage:

- Windows: pepperStart.bat -p workflow-file
- Unix/Linux/MacOS: bash pepperStart.sh -p workflow-file

The content of the workflow-file is described in the following section.

6.3 Pepper Workflow

The worklfow of a conversion process in Pepper consists of three phases: An import phase, a manipulation phase and an export phase.

- In the import phase, modules (called importers) map data from an input format to Salt, the metamodel used to describe all types of data.
- In the manipulation phase, modules (called manipulators) map data from one Salt model to another Salt model (to alter data e.g. by renaming certain annotation names).
- In the export phase, modules (called exporters) map data from Salt to an export format.

Each phase can include several steps. The export-phase and the import-phase can include 1 to n steps, whereas the manipulation-phase can include 0 to n steps. Steps are the lifecycles of running a module (i.e. a PepperModule). Every module can be identified by a name (the module-name). In addition, importers and exporters also can be identified by a pair consisting of the format name and the format version they support. During

processing, Pepper searches for a module with a given module name (or a given pair of format name and format version) and starts it. Additionaly for every module you can add a file with parameters for this module. Please see the description of the module you want to use for details. Importers as well as exporters also needs a path to the file or path they are supposed to import from or export to.

Modeling a Workflow via XML:
An xml file defining a module is called a Pepper-workflow file and has the ending ".pepperparams". A workflow description (using module names for identification) looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<PepperParams:PepperParams xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:PepperParams="de.hub.corpling.Pepper.pepperParams">

    <PepperJobParams id="1">

            <importerParams moduleName="" sourcePath=""
specialParams=""/>

            <!-- ... -->

            <moduleParams moduleName=""  specialParams=""/>

            <!-- ... -->

            <exporterParams moduleName="" destinationPath=""
specialParams=""/>

            <!-- ... -->

    </PepperJobParams>

    <!-- ... -->

</PepperParams:PepperParams>
```

The xml-element „PepperJobParams" stands for a Pepper job. One job does one conversion (you can specify one or more jobs in one workflow file). Every job has to have a unique id and has to contain at least one importer description and one exporter description. A manipulator description is optional. There is no upper limit for the number of module descriptions which can be used for a conversion. The attribute „moduleName" identifies the module which is to be used for the current step. Importers have an attribute „sourcePath", where you have to specify the path of the source corpus. Exporters have an attribute „destinationPath" where you have to specify the path of the destination corpus. The attribute „specialParams" can be used for parameters for the current module. SpecialParameters must be given in a property file.

Caution: Please make sure that every path is in URI-syntax and is an absolute path.
A workflow description (using format name and format version for identification of im- and exporters) looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<PepperParams:PepperParams xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:PepperParams="de.hub.corpling.Pepper.pepperParams">

    <PepperJobParams id="1">

            <importerParams formatName="" formatVersion=""
sourcePath="" specialParams=""/>

            <!-- ... -->

            <moduleParams moduleName=""  specialParams=""/>

            <!-- ... -->

            <exporterParams formatName="" formatVersion=""
sourcePath="" destinationPath="" specialParams=""/>

            <!-- ... -->

    </PepperJobParams>

    <!-- ... -->

</PepperParams:PepperParams>
```

Unlike the upper example here we use the attributes "formatName" and "formatVersion" to identify an importer as well as an exporter.

6.4 Example

In PEPPER_HOME you will find a folder *examples* with a small sample corpus for conversion (this is the pcc2 demo corpus in the PAULA XML format). The following workflow-file defines the conversion of this corpus from PAULA to the relANNIS format.

```
<?xml version="1.0" encoding="UTF-8"?>

<PepperParams:PepperParams xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:PepperParams="de.hub.corpling.Pepper.pepperParams">

    <PepperJobParams id="1">
```

```
            <importerParams moduleName="PAULAImporter"
sourcePath="file:/PEPPER_HOME/examples/sample1/paula/pcc2/" />

            <exporterParams moduleName="RelANNISExporter"
destinationPath="file:/PEPPER_HOME/examples/sample1/relANNIS/"/>

    </PepperJobParams>

</PepperParams:PepperParams>
```

This file also can be found under PEPPER_HOME/examples/sample1/ paula2relANNIS.pepperParams. For testing, you can call:

```
 pepperStart.bat -p
PEPPER_HOME/examples/sample1/paula2relANNIS.pepperParams
```

or

```
 bash pepperStart.sh -p
PEPPER_HOME/examples/sample1/paula2relANNIS.pepperParams
```

Take care to replace PEPPER_HOME with the absolute path of the pepper-directory.

After doing this you will find the newly created folder "relANNIS" in PEPPER_HOME/examples/sample1/relANNIS/ which contains the pcc2-corpus in the relANNIS-format. The following example will show a similar workflow producing exactly the same result, but here instead of identifying the PepperModule by using the name, we use the format name and the format version:

```
<?xml version="1.0" encoding="UTF-8"?>

<PepperParams:PepperParams xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:PepperParams="de.hub.corpling.Pepper.pepperParams">

    <PepperJobParams id="1">

            <importerParams formatName="PAULA"
formatVersion="1.0"
sourcePath="file:/PEPPER_HOME/examples/sample1/paula/pcc2/" />

            <exporterParams formatName="relANNIS"
formatVersion="3.0"
destinationPath="file:/PEPPER_HOME/examples/sample1/relANNIS/"/>

    </PepperJobParams>

</PepperParams:PepperParams>
```