D-SPIN Joint Deliverables

R5.3: Documentation of the Web Services

R5.4: Final Report

March 2011

D-SPIN, BMBF-FKZ: 01UG0801A

Deliverables:

R5.3: Documentation of the Web Services

R5.4: Final Report WP5

Responsible: Erhard Hinrichs

Editors: Erhard Hinrichs, Kathrin Beck, Wolfgang Maier

Summary

D-SPIN is the German counterpart of the European Research Infrastructure project CLARIN (Common Language Resources and Technology Infrastructure, http://www.clarin.eu). The ultimate objective of CLARIN is to create a European federation of existing digital repositories that include language-based data, to provide uniform access to the data, wherever it is, and to provide existing language and speech technology tools as web services to retrieve, manipulate, enhance, explore and exploit that data. The primary target audience is researchers in the humanities and social sciences and the aim is to cover all languages relevant for the user community.

Similar goals are pursued by the D-SPIN project on the national level. Work in D-SPIN is carried out in close collaboration with CLARIN. Within the CLARIN federation, the focus of D-SPIN is on German resources, tools and their integration through web services. Besides these localization efforts, D-SPIN has a special focus in addressing potential users of the infrastructure with the preparation of training material and teaching activities.

This document is a report of Work Package 5 "Language Resources and Tools". The first part of the document presents the integration of language resources of the project partners within the CLARIN project. The second part is a documentation of WebLicht, the web services platform implemented by the project partners, which integrates tools and resources. The documentation shall facilitate its use for its current and future users. Apart from a general description of WebLicht, and tutorials intended for WebLicht users, several particular technical aspects are documented. These include a detailed description of the internal structure of WebLicht, a description of the data exchange format TCF, and a description of the different types of tools offered in WebLicht.

1 Introduction

The task of WP5 was to identify language resources of the partners and of further interested sites in Germany and to prepare the integration of these resources and tools into the CLARIN/D-SPIN LRT federation.

In close collaboration with the CLARIN project, the following has been accomplished.

Integration of language resources

Partners have registered their resources with the resource repository on the CLARIN website and have started to convert resources to facilitate the integration. Details of this process are presented in section 2.

Integration of tools and resources via web services

Four partners (BBAW, University of Leipzig, University of Stuttgart, University of Tübingen) have implemented and deployed the provided web services for

- The access to lexical resources, e.g. "GermaNet", "Deutscher Wortschatz" and "Wörterbuch der deutschen Gegenwartssprache" and
- The processing and linguistic annotation of text (amongst other things for tokenization, sentence boundary detection, part-of-speech tagging, parsing, named entity recognition, base form reduction, co-occurrence annotation and semantic role annotation).

The resources and tools have been combined into complex workflows. TCF, a format for the interchange of textual data between these tools has been specified. A GUI for the combination of web services, WebLicht,[1] has been implemented. Section 3 gives a general project description of the WebLicht. Section 4 is dedicated to a presentation of the D-SPIN web service registry. Section 5 explains the WebLicht chaining logic. Section 6 presents TCF, the data exchange format used within WebLicht. Section 7 explains the approach for data visualization in WebLicht. Section 8 contains an explanation of the different types of tools offered by WebLicht. Sections 9 is concerned with services resulting from collaboration efforts between project partners (project C4, BBAW, international partners). Eventually, section 10 contains tutorials on how to integrate web services into the WebLicht architecture.

2 Integration of Language Resources

Partners have registered their resources with the resource repository on the CLARIN website and have started to convert resources to facilitate the integration. The resource repository, called CLARIN Virtual Language Observatory (CLARIN VLO)[2] provides convenient web-based access for everyone. Among other tools, the VLO offers a resource browser.[3] A list of resources that have been registered by project partners as of Friday, January 28, 2011, is given in

[1] Available at https://weblicht.sfs.uni-tuebingen.de

[2] Available at http://www.clarin.eu/vlo/

[3] Available at http://www.clarin.eu/view_resources/

Appendix C.

3 General Description of WebLicht

3.1 Description of the Project

WebLicht (Web Based Linguistic Chaining Tool, see http://www.d-spin.org for an overview) is a web-based service environment for the integration and use of language resources and tools. It includes NLP tools (realized as web services), a repository for storing relevant information about each service, and a chaining mechanism that determines how the tools can be combined. These elements are brought together in a web application that enables users to create and run tool chains on their own texts.

WebLicht is in an advanced prototypical state and is already used in a productive way.

3.2 Bodies Responsible

The WebLicht infrastructure was designed and implemented in a collaborative effort by the following institutions:

- Department of Linguistics, University of Tübingen (Uni-Tübingen), Germany
- Institute for Natural Language Processing, University of Stuttgart (Uni-Stuttgart), Germany
- Natural Language Processing, University of Leipzig (Uni-Leipzig), Germany
- Berlin-Brandenburgische Akademie der Wissenschaften (BBAW), Berlin, Germany
- Institut für Deutsche Sprache (IDS), Mannheim, Germany

Other institutions and individuals have supported the project by providing tools in the form of web services, including:

- Department of Modern Languages, University of Helsinki, Finland
- Romanian Academy Research Institute for Artificial Intelligence (RACAI), Bucharest, Romania
- Institute of Computer Science, Polish Academy of Sciences (PAN), Warsaw, Poland

3.3 Motivation for the Project

WebLicht was developed to alleviate the problems associated with download-first tools and to provide a means for distributed tools to be combined into service chains. At present, digital language resources and tools (LRT) are available in different forms: some of them can be downloaded from the worldwide web, while others are shipped on CD-ROMs or DVDs. These tools are often restricted to a particular platform or environment, but some can be used independently of operating systems or data formats. In either case, the software must be installed or copied to a local computer or network. This can lead to problems that may be difficult to resolve. For example:

- The users' machines are often not able to handle the needed resources or tools. This includes incompatibilities between libraries and operating systems as well as the fact that even modern PCs are sometimes not powerful enough to fulfill the requirements of linguistic analysis.
- The lack of common data formats makes it impossible to create chains of tools or to compare different resources with each other easily.

In recent years, many attempts were made to find solutions to these persistent issues. Standardized text encodings like Unicode and markup languages like XML are addressing the lack of common data formats, while platform independent programming languages like Java and Python provide a

promising way to make tools available across platforms. But still, the user must rely on the capabilities of his local machine.

To overcome this restriction, WebLicht makes the functionality of linguistic tools and the resources themselves available via the Internet. To make use of the more than 120 tools and resources currently in WebLicht, the end user needs nothing more than a common web browser. The tools share a common data format, which means that any tool can make use of annotations that were produced earlier in the chain.

3.4 Supported Languages

There are currently services available for German, English, Italian, French, Spanish, Romanian, Finnish, Czech, Hungarian, and Slovenian. However, the WebLicht infrastructure itself does not restrict the languages supported. A language is considered "supported" when one or more tools for processing it are available.

3.5 Implemented NLP Web Services, Availability

Approximately 120 web services are available, including:

- Tokenizers
- Sentence border detection
- Part-of-speech taggers
- Named entity recognition
- Lemmatization
- Constituent parsing
- Coocurrence annotation
- Semantic annotator (GermaNet)
- Data format converters (including MS Word/PDF/RTF to TCF, plain text to TCF, TCF <-> MAF, Negra, TEI)

Available at https://weblicht.sfs.uni-tuebingen.de (login via Shibboleth required)

3.6 WebLicht Architecture

In WebLicht, tools and resources are implemented as distributed services in a Service-Oriented Architecture (SOA) (Henrich, Hinrichs, Hinrichs, & Zastrow, 2010). That means that they are not running on one central machine, but instead they are running on many different machines that are distributed across the web. A centralized database, the repository, stores technical and content-related metadata about each service. With the help of this repository, the chaining mechanism is implemented. The WebLicht user interface encapsulates this chaining mechanism in an AJAX-driven web application. It provides ways of uploading or generating a text corpus and guides the user in creating tool chains. Since web applications can be invoked from any browser, downloading and installation of individual tools on the user's local computer is avoided.

3.6.1 Service-Oriented Architecture

In most cases, the services are running at the institute where the tool was originally developed. This allows the authors to maintain control over their services.

Distributing the services in this way has several advantages. The computational workload is spread among various machines, resulting in better performance. It also has the advantage that when a tool is improved, the new version is immediately available. End users do not need to do anything to take advantage of the improved tool.

The services are created as *RESTful web services*, which means that they make use of the HTTP protocol. Services do not interact directly with each other, but are invoked individually by sending the required document via HTTP.

3.6.2 Repository

To be available in WebLicht, a service must be registered in a centralized repository. The repository stores two different kinds of information about each service:

- Technical metadata: they provide details about what is required in the input document and what output the service generates. This information is needed to be able to compute valid tool chains. For example, a POS tagger normally needs a tokenizer to be applied to the text before it can be invoked.
- Descriptive metadata: concerning author, location of the service, legal information, description of the service etc.

The repository itself can be queried by invoking RESTful web services, thus making it possible for both the chaining mechanism and the web GUI to retrieve information about the services available in WebLicht.

3.6.3 Chaining Mechanism

The chaining mechanism determines which processing chains can be formed. This is done by matching the input requirements of the services in the repository with the data available in the document. The chaining process can be applied prior to actually invoking service tools, which makes it possible to build entire service chains quickly. This is achieved by creating a chain "profile" which is updated each time a service is added to the chain. The updated profile is used at each step to determine which services can possibly follow in the chain.

3.6.4 Interoperability

An important part of Service Oriented Architectures is ensuring interoperability between the underlying services. Interoperability of web services, as they are implemented in WebLicht, refers to the seamless flow of data through the web services in the processing chain. To be interoperable, these web services must first agree on protocols defining the interaction between the services (WSDL/SOAP, REST, XML-RPC). They must also use a shared data exchange format, which is preferably based on widely accepted formats already in use (UTF-8, XML). WebLicht uses the REST-style API and a data format (Text Corpus Format, TCF) that is a valid XML format fully compliant with the Linguistic Annotation Format (LAF) and Graph-based Format for Linguistic Annotations (GrAF) developed in the ISO/TC37/SC4 technical committee (Ide & Suderman, 2007). The TCF format allows the combination of the various linguistic annotations produced by the tool chain. It supports incremental enrichment of linguistic annotations at different levels of analysis in a common XML-based format. Thus, the web services within WebLicht are highly interoperable. Although TCF is used by most web services, it is also possible to use other formats. Currently, converters to and from TCF and MAF, TEI and Negra are available or in development.

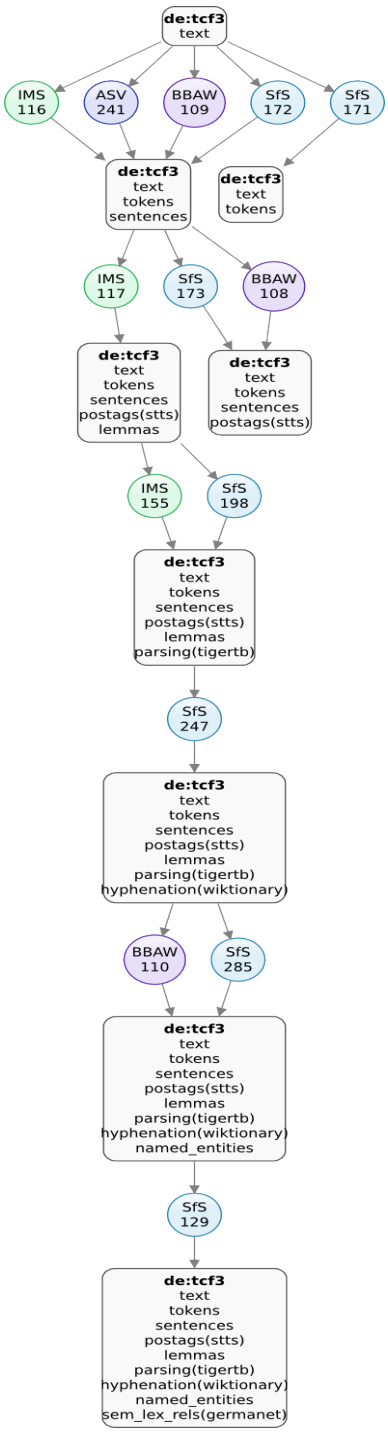


Figure 1: Subset of web services available for German

3.6.5 Sample Service Chains

WebLicht architecture allows a high degree of flexibility with respect to service processing chains. For example, Figure 1 shows a small subset of possible processing chains available for the German language. The circles represent individual web services (labeled with the institute which created it and an identification number) and the rectangular boxes represent linguistic annotation states. An annotation state indicates which linguistic annotation layers are present in the data document. Often it is possible to reach an annotation state through different processing chains. This allows one to compare results from different tools and to pick-and-choose those tools that the user deems best suited to their needs.

Figure 2 shows the connectivity of all of the web services currently available in WebLicht for the German language.

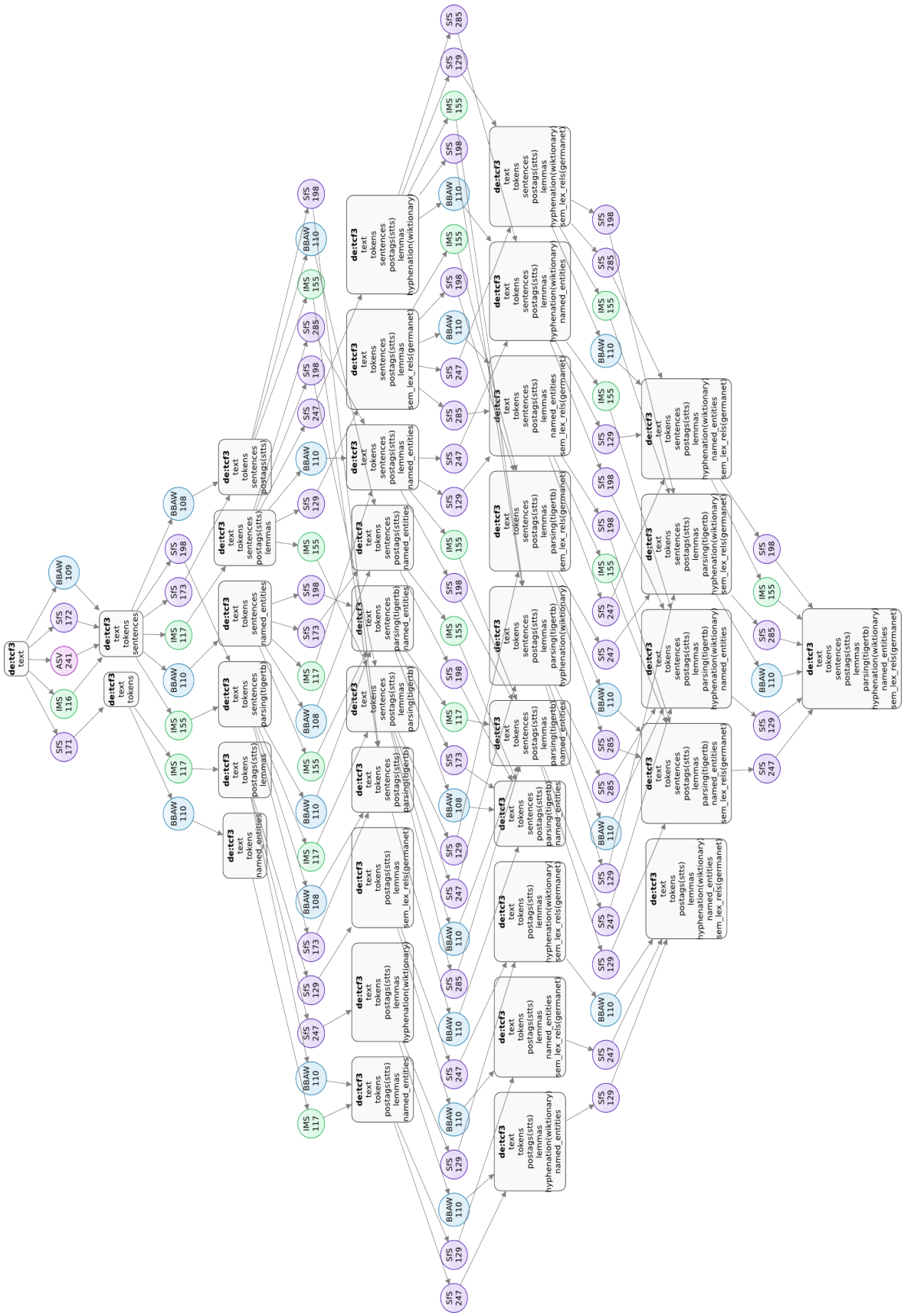


Figure 2: Complete Set of Web Services for German

4 Description of the D-SPIN Web Service Registry

The D-SPIN web service registry manages metadata on web services of the WebLicht infrastructure. It was developed and is maintained by the D-SPIN team in Leipzig. The web service registry allows the registration of new services, modification of the metadata of already registered services and the deregistration of services. The registry is located on the persistence layer of the WebLicht infrastructure and therefore is one of the basic components of the architecture.

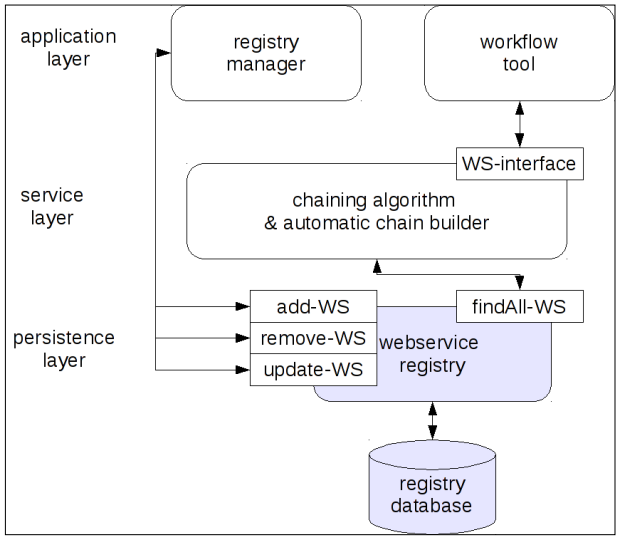


Figure 3: The registry as part of the persistence layer

The metadata that is managed can be divided into two different kinds:

- Technical metadata: orchestration and invocation metadata used by the infrastructure
- Non-technical metadata: metadata interesting to a human user / scientist

Orchestration metadata is needed in order for the chaining logic to work. Basically this metadata provides an interface description on the inputs and outputs of a service. Additionally information on how a service can be invoked and an URL, that specifies where the service is located at, is stored. This technical metadata usually is of no interest to the human user, but is needed by other components of the WebLicht infrastructure. Summing this up, the following technical metadata is stored for each service:

Input format	A unique identifier specifying the input format consumed by a service.
Input version	A string defining the version of the input format that is consumed.
Input parameters	A set of triplets (layer, attribute, value) of unique identifiers specifying the parameters that need to be present in the input in order for a service to work.
Output format	A unique identifier specifying the output format produced by a service.
Output version	A string defining the version of the format of the output that is produced.
Output values	A set of triplets of unique identifiers specifying the results added to the input on invocation of a service.
URL	A URL pointing to the location of the service.

Table 1: Technical metadata - overview

Input and output values are defined as triplets of unique identifiers. These triplets consist of:

- A unique identifier specifying a layer, e.g. text, token, POS-tags
- A unique identifier specifying an attribute that further defines a certain aspect of a layer, e.g. tag set
- A unique identifier specifying the value of the attribute, e.g. STTS[4]

Please note: In case there are no attributes for a layer, the second (attribute) and third (value) part of the triplet may be empty.

In contrast to technical metadata, non-technical metadata provides additional information on the service that is of interest to a human user. This includes the following key-value pairs:

Name	The name of the service.
Description	A detailed description of the service.
Short description	A short description of the service.
Creator	The creator of the service.
Contact	Information on who/how to contact (usually a mail address of the creator).
Status	The status of the service, which may be one out of {development, production, deprecated}.

Table 2: Non-technical metadata - overview

Additionally the repository automatically manages the following information:

ID	A unique identifier for a service managed by the repository.
Password	A hash of a password provided upon registration and checked for consistency every time the registry deals with modification/deletion requests.
Created	A timestamp specifying the date a service was first registered to the registry
Modified	A timestamp specifying the latest date of modification of the metadata of a service.

Table 3: Metadata created and controlled by the registry - overview

The web service registry can be accessed using REST-web services. Services are registered, modified and deleted using HTTP-POST commands, while read operations are invoked by HTTP-GET commands. To keep the documentation short we will omit full paths and only specify the interesting final parts of an URL. A call to */RepositoryFindAll means that we call http://registryHost:registryPort/somePath/RepositoryFindAll.

A HTTP-GET request to */RepositoryFindAll is answered by a XML document containing metadata on all services known to the registry. Services are managed by sending the following HTTP-POST requests:

- */RepositoryAdd: Adds a new service to the registry using the provided metadata

4 Stuttgart Tübingen Tagset. See (Schiller, Teufel, & Thielen, 1995) and http://www.sfb441.uni-tuebingen.de/a5/codii/info-stts-en.xhtml

- */RepositoryDel: Removes a service from the registry in case a valid id and password are provided

Since the metadata information possibly exceeds the length of a common URL (specifications differ, but common sense seems to be a maximum length of 256 or 1024), all data is provided in an XML document sent via HTTP-POST. Registration and modifications are done using */RepositoryAdd and specifying the id and password of a service already registered to the registry. Due to the prototypical state of the implementation, some convenience functionality needed in order for the registry interface to be 100% restful was omitted. For example it's not possible to address a single service description or certain parts of it, like the service name/url/..., using the common REST-style (*/Repository/Service/268/url). Since this was not an important feature in a prototypical implementation of the infrastructure, this convenience functionality was not added.

The registry uses a dedicated XML format when communicating via web services. Usually a list of services or a single service is described. The metadata is encapsulated into a service tag. Its schema reflect the different types of information that can be stored in the registry:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<CLARIN>
 <service id="268">
  <contact>wort@informatik.uni-leipzig.de</contact>
  <creator>ASV Universitaet Leipzig</creator>
  <description>Transforms all tokens from TextCorpus to lemmas in Lexicon</description>
  <shortdescription>transforms TextCorpus to Lexicon</shortdescription>
  <name>ULei - TextCorpus2Lexicon</name>
  <url>http://myhost:myport/somePath/TextCorpus2Lexicon?language=es</url>
  <status>production</status>
  <created>2010-08-23 14:57:42.0</created>
  <modified>2010-09-23 08:16:22.0</modified>
  <inputwrapper>textcorpus,0.3</inputwrapper>
  <outputwrapper>dspin.pid.wrapper.dspinlexicon</outputwrapper>
  <inputs lang="es" type="textcorpus,0.3" version="0.3">
   <layer name="lang"><attr name="languagecode" value="es"/></layer>
   <layer name="tokens"/>
  </inputs>
  <outputs lang="es" type="dspin.pid.wrapper.dspinlexicon" version="">
   <layer name="lemma"/>
   <layer name="lang"><attr name="languagecode" value="es"/></layer>
  </outputs>
 </service>
<CLARIN>
```

Snippet 1: Registry XML format - example

In case multiple services are returned, for example when computing a findAll-request, the different services are encapsulated into a services tag, containing a list of services.

The back-end of the registry consists of a database that is managed in a MySQL[5] database system. The web services interface is provided via Java Servlets[6] running in an Apache Tomcat[7] instance. Since the registry is a very crucial part of the WebLicht infrastructure, a simple backup and mirroring strategy was implemented in order ensure its availability. The database itself is mirrored to a backup MySQL database system every 24 hours. There are two Apache Tomcat instances running on different machines. In case one cannot be reached, the second one can be used by other infrastructure components. Both instances first try to access the main database. If this one cannot be reached, the database mirror is contacted automatically. In this case write access, like registering

5 http://www.mysql.de

6 http://www.oracle.com/technetwork/java/javaee/jsp/index.html

7 http://tomcat.apache.org/

new services or modifications on existing ones, are forbidden. But read-access to the registry is still possible, allowing the chaining component and other components like the WebLicht user interface to work.

Although the registry can be accessed via web services, human users prefer using a graphical user interface to creating XML-documents and issuing web service requests manually. Therefore a simple registry management tool was implemented by the team in Leipzig. It provides simplified access to the repository and supports the user especially when registering new services or modifying existing ones. Upon registration of a new service only valid layer, attribute and value combinations are available to the user. Due to security reasons this tool is currently not available to non-members of the D-SPIN infrastructure.

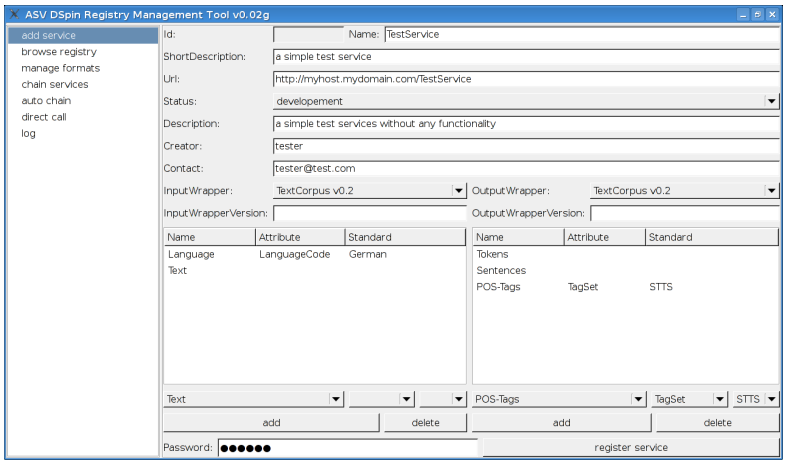


Figure 4: Registry Management Tool

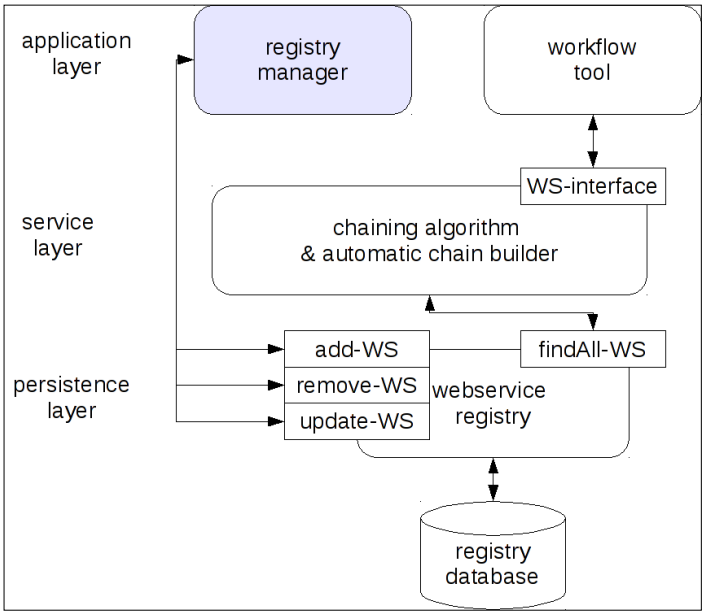


Figure 5: The Registry Management Tool as part of the application layer

5 Description of the Chaining Logic

One of the basic ideas behind WebLicht is to not only offer a certain amount of linguistic web services providing access to data and tools, but to enable users to easily define workflows built out of interchangeable building blocks. Therefore one of the basic principles and goals of the WebLicht architecture is to allow the automatic and semiautomatic orchestration of workflows.

A workflow is defined by a set of web services. Upon invocation, these web services are invoked one after another, the output of one service in the chain being used as input of the next one. This chaining of web services can be compared to the nesting of functions: The result of function f_a is used as input of function f_b, its result is consumed by a function f_c and so on: $f_c(f_b(f_a))$.

This nesting of functions for example is commonly used in programming languages. In this comparison a programmer acts as an orchestrator, and a type checker checks on compile time if a certain nesting is allowed. This check is based on the compatibility of the types being used in the signature (interface specification) of the involved functions/methods.

This process is reflected by the chaining logic used in the WebLicht chaining algorithm. Service descriptions are stored and handled by the WebLicht service registry and the provided technical metadata is based on this concept. For example a NLP web service, lets assume it's a POS-tagger for German text, in WebLicht produces and possibly consumes:

1. A document formatted according to a defined standard, for example D-SPIN-TextCorpus/TCF[8]
2. A document containing a certain set of types of information, for example POS-tags,...
3. … which are either just present or also encoded using a specified standard, for example STTS[9] for POS-tagging in German.

In short: A web service description of a WebLicht service consists of input and output definitions, specifying the types of data that are consumed and produced. Additionally the following assumptions are made:

- A web service consumes a document that is well formed according to a certain standard
- A web service produces a document that is well formed according to a certain standard that can be different from the type of the input document
- If a web service produces and consumes a document of the same type, it may add new types of information to the document or modify existing ones, but does not remove any kind of information.

Please note that the chaining algorithm itself is independent from the format of the documents being usually chained in WebLicht. In fact it is even independent from the fact that XML-documents, or something completely else, are interchanged. Of course, most of the services in WebLicht rely on the TCF format. But this is only in order to start off with a huge set of compatible services, but no issue of the chaining logic itself.

---

8 A corpus representation format for linguistic web services used in D-Spin (Heid, Schmid, Eckart, & Hinrichs, 2010)

9 Stuttgart Tübingen Tagset. See (Schiller, Teufel, & Thielen, 1995) and http://www.sfb441.uni-tuebingen.de/a5/codii/info-stts-en.xhtml

Table 4 and Table 5 provide an example of complete, WebLicht compatible input- and output descriptions of a POS-tagger web service. A document in TextCorpus-format is consumed. This document must contain tokenized German text encoded in utf8. The service adds POS-tags according to the STTS standard and returns the resulting document.

Format	TextCorpus
Input	text = utf8 language = German tokens = present

Table 4: Example input specification for a POS-tagger web service

Format	TextCorpus
Output	POS-tags = STTS

Table 5: Example output specification for a POS-tagger web service

The basic idea of the chaining logic is to administrate a document profile, representing the state of the document being handled at every step of a chain. In order to decide whether a certain service is compatible and may be invoked on this document, the documents profile is compared to the input description of the service. If every feature that is present in the input description of the service is present in the document profile, the service is compatible and can be invoked. After invocation of the service the features present in the output specification of the service are added to the profile of the handled document and the process may start again. A graphical representation of the invocation of the service previously described in Table 4 and Table 5 is shown in Figure 6.

The process described above can also be applied on built time of a chain, without actually invoking any service. The profile of a virtual document is handled and a chain built step by step, matching the virtual profile against available services.

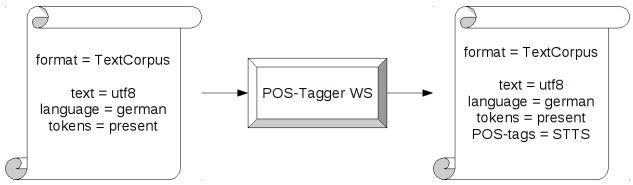


Figure 6: Invocation of a POS-tagger web service

A more formal definition of the chaining logic is: A service WS_n is compatible to a chain of services $\{WS_1,\ldots,WS_{n-1}\}$, if the two following constraints are fulfilled:

- The format in the output of the predecessor service WS_{n-1} is equal to the format defined in the input specification of WS_n.
- All features out of the input specification of WS_n are present in the profile that was created by all previous services in the chain $\{WS_1,\ldots,WS_{n-1}\}$.

By using the WebLicht user interface, which itself relies on the chaining algorithm, human users are able to build chains semi-automatically. Compatible services are presented to the user. The user picks one of these services and adds it to the end of a chain. This process ends if no compatible services are found or the user decides to invoke the current chain.

Based on this simple chaining logic, an automatic chain builder can be implemented. This was done on an experimental level. The basic idea is, that an inexperienced user can be supported in building workflows by automatically finding chains only constrained by user defined starting and end points of a chain:

- Starting point: German text
- End point: part of speech tagged text; used tag set should be STTS

Starting and end points of such a chain can be represented via concrete services. Therefore the first functionality needed is a service that allows some kind of search over all existing services. For example a search for STTS may return services encapsulating a POS-tagger producing STTS POS-tags. Once both the start and end of a chain are defined through a service, an automatic method that calculates one ore more chains fulfilling these two constrains is needed. An experimental implementation of this functionality is already available.

A basic formalization of the automatic chain builder algorithm looks like this:

- W_{start} is the starting point service and W_{end} the end point service
- L_{all} is the list of all services, also containing W_{start} and W_{end}

```
  o  Create an empty list of service chains L_c
  o  Create a one element chain C = {W_start}
  o  Copy all elements of L_all to an previously empty list of
     services L_s
  o  Remove all services from L_s that are not compatible to the
     last entry in C, according to the chaining algorithm.
  o  Remove all services from L_s that are already present in C
     // prevent circles
  o  For each entry S in L_s:
       Create a copy C_2 of C and add S to the end of C_2

       if S = W_end:

            yes: add C_2 to L_c

            no: recursively go back to 3) and set C = C_2
```

Snippet 2: Sketch of the automatic chain builder algorithm

There are some more problems like circles (chains containing the same service more than once), aggregates (a service encapsulating the functionality of other services) and doublets (chains that only

differ because of reordering of the involved services) that were also addressed and partly/pragmatically solved. For further information concerning the chaining logic used in WebLicht and the automatic chain-building algorithm, we refer to the according LREC paper (Boehlke, 2010).

Both, the chaining algorithm and the automatic chain-building algorithm, are located at the service layer of the D-SPIN infrastructure. The automatic chain builder makes use of the chaining algorithm and both are available via REST web services.

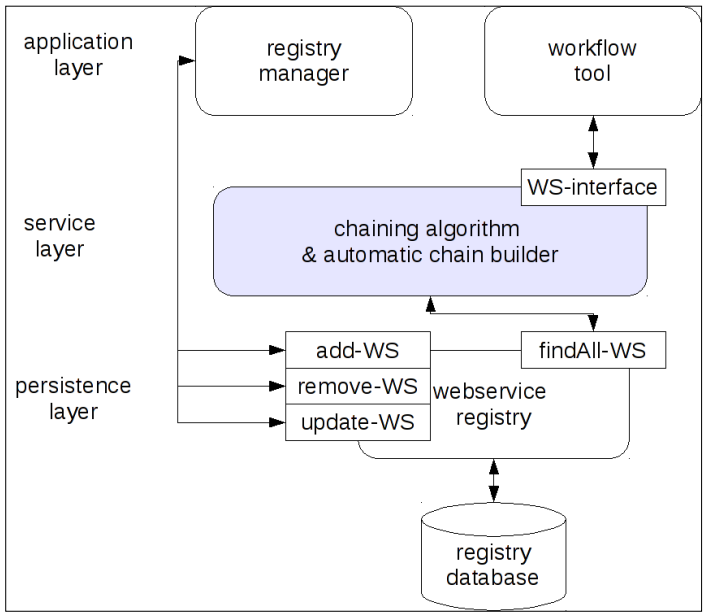


Figure 7: The chaining algorithm and automatic chain builder as part of the service layer

In order to keep the documentation short we will again omit full paths and only specify the interesting final parts of an URL. The chaining algorithm is accessed by sending HTTP-GET requests to */RepositoryChain. A chain of services can be provided using the URL-parameter serviceIds. The IDs of services known to the repository the chainer is running upon need to be used. Let id=1 be a service that converts a given binary file containing text to a TCF-file and let id=2 be a service that offers sentence segmentation and tokenization on the text present in a TCF-document. In case */RepositoryChain?serviceIds=1,2 is called, the chainer will return services compatible/working on a TCF-document containing tokenized text. For example this may be a POS-tagger. The repositoryChain-service returns an XML-document, which contains a list of compatible services. Its format is similar to the one used by the registry findAll-service (see Snippet 2), additionally adding a services-tag that aggregates several services to a list/chain of services (see Snippet 3). This may also be an empty list, if no compatible services are found.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<CLARIN>
 <services>
  <service id="1">
   <contact>wort@informatik.uni-leipzig.de</contact>
   ...
  </service>
  <service id="2">
   ...
  </service>
  ...
 </services>
```

```
</CLARIN>
```

Snippet 3: Chaining algorithm - XML output, shortened example

The automatic chain builder can be accessed by sending HTTP-GET requests to */RepositoryFindChain. Similar to the chaining algorithm interface, again a list of service ids specifying a chain, which may of course also consist of only one service, has to be defined using the serviceIds-parameter. Optionally a goal, consisting out of a single service, for the chain(s) to be built automatically can be specified using the parameter targetId. If a goal is specified, all possible chains starting with the chain defined through the serviceIds-parameter and ending with the service defined by targetId are returned. If targetId is omitted all possible chains, starting with the chain specified by serviceIds up to the point where no compatible service can be found that isn't already part of the chain, are returned.

Examples:

- RepositoryFindChain?serviceIds=1 : returns all possible, maximized (no further compatible service not already present in the chain can be added to the end of the chain) chains starting with the chain {1}
- RepositoryFindChain?serviceIds=1,2 : returns all possible, maximized chains starting with the chain {1,2}
- RepositoryFindChain?serviceIds=1,2?targetId=4 : returns all possible, maximized chains starting with the chain {1,2} and ending with service 4 (for example {1,2,3,4})

Additionally the maxChains-parameter may be used. In case a positive integer value is defined, the top x shortest chains matching the constraints defined by the serviceIds- and targetId-parameter are returned. If maxChains is omitted, the automatic chain builder is invoked using the default maxChains=10.

Examples:

- RepositoryFindChain?serviceIds=1,2?maxChains=50 : returns the 50 shortest chains starting with the chain {1,2}
- RepositoryFindChain?serviceIds=1,2?targetId=4?maxChains=5 : returns the 5 shortest chains starting with the chain {1,2} and ending with service 4

The automatic chain builder returns XML-documents similar to the one of the chaining algorithm (see Snippet 3). But in this case, multiple chains are defined by several occurrences of the services-tags.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<CLARIN>
 <services>
  <service id="1">
   ...
  </service>
  ...
 </services>
 <services>
  <service id="2">
   ...
  </service>
  ...
 </services>
 ...
</CLARIN>
```

Snippet 4: Automatic chain builder - XML output, shortened example

Currently different enhancements to the chaining logic are discussed in the D-SPIN community. A main issue is to further widen the expressiveness of the service descriptions stored in the WebLicht service registry. Since this is closely related to the discussion on the orchestrations specific parts of CMDI based web service descriptions in CLARIN, the D-SPIN team works in close collaboration to other CLARIN partners on that issue.

6 Description of the TCF Format

6.1 Introduction

The **TCF** data format ("ToolChaining Format") is the primary data interchange format used by D-SPIN web services and was jointly developed by the D-SPIN partners. TCF is an **XML-based format** that is encoded in **Unicode**. A typical TCF document contains a *text corpus* and a set of *linguistic annotations* such as tokens, part-of-speech (POS) tags, lemmas, or syntactic structures. TCF encodes each type of annotation in a separate *layer*.

A linguistic processing chain often starts with a raw corpus that only consists of a text. The corpus is sent through a pipeline of web services. Each web service adds one or more new annotation layers, and the enriched corpus is sent to the next web service of the processing chain for further annotation. Such processing chains can be composed with the WebLicht tool.

The following sections describe the TCF format in detail, and the appendix contains RelaxNG schemas for version 0.4 of the TCF format.

6.2 The "TextCorpus" Element

6.2.1 The "text" Layer

Here is a very simple TCF document[10].

```
<D-Spin version="0.4">
   <TextCorpus lang="de">
      <text>Peter aß eine Pizza. Sie schmeckte ihm.</text>
   </TextCorpus>
<D-Spin>
```

It contains a short text inside a <text> element, which is embedded in a <TextCorpus> element, which in turn is embedded in the top element <D-Spin>. The <D-Spin> element has an attribute "version" which specifies the version of the TCF format used for this document. The current version is 0.4 (as of February 2011). The <D-Spin> element has an optional attribute "lang" specifying the language of the corpus.

6.2.2 The "tokens" Layer

Linguistic annotations are encoded by further XML elements that are added at the same level as the <text> element, i.e. below the <TextCorpus> element. The following TCF document contains a *tokens* layer:

[10] This example is slightly simplified because it lacks name spaces and the obligatory <MetaData> element.

```
<D-Spin>
  <TextCorpuslang="de">
    <text>Peter aß eine Salamipizza. Sie schmeckte ihm.</text>
    <tokens charOffsets="true">
      <token ID="t1" start="0" end="4">Peter</token>
      <token ID="t2" start="6" end="7">aß</token>
      <token ID="t3" start="9" end="12">eine</token>
      <token ID="t4" start="14" end="24">Salamipizza</token>
      <token ID="t5" start="25" end="25">.</token>
      <token ID="t6" start="27" end="29">Sie</token>
      <token ID="t7" start="31" end="39">schmeckte</token>
      <token ID="t8" start="41" end="43">ihm</token>
      <token ID="t9" start="44" end="44">.</token>
    </tokens>
  </TextCorpus>
</D-Spin>
```

Each token is represented by a <token> element that encloses the string of the token. The <token> element has an optional “ID” attribute that uniquely identifies it. It may also have “start” and “end” attributes to indicate the start and end position of the token in the character string of the <text> element. The <token> elements are embedded inside of a <tokens> element. The optional Boolean “charOffsets” attribute of the <tokens> element indicates that start and end attributes are available at the <token> elements if its value is true.

6.2.3 The “sentences” Layer

The segmentation of a text into sentences is represented with <sentence> elements. Each <sentence> element has a “tokenIDs” attribute that specifies the set of tokens that the sentence contains. The tokens are referred to by their IDs. Just like <token> elements, the <sentence> elements also have optional “ID” and “start” and “end” attributes. All <sentence> elements are enclosed in a <sentences> element. The “charOffset” attribute again indicates whether “start” and “end” attributes are present at the <sentence> elements.

The following partial TCF document shows how sentence boundary information is added to the previous document:

```
...
    </tokens>
    <sentences charOffsets="true">
      <sentence ID="s1" tokenIDs="t1 t2 t3 t4 t5" start="0" end="25"/>
      <sentence ID="s2" tokenIDs="t6 t7 t8 t9" start="27" end="44"/>
    </sentences>
  </TextCorpus>
</D-Spin>
```

6.2.4 The “POStags” Layer

Part-of-speech (POS) information is encoded in the *POStags* layer. The <POStags> element has an attribute “tagset” which specifies the tag set from which the inventory of POS tags is taken. The <POStags> element contains a sequence of <tag> elements that enclose the POS labels.

Each <tag> element has an attribute “tokenIDs” which specifies the token(s) to which the POS tag is assigned. It is possible to assign a single POS tag to a sequence of tokens. The token sequence may even be discontinuous which allows particle verbs such as “fährt ... ab” in “Der Zug fährt gleich ab” to be assigned a single POS tag.

It is also possible that two different <tag> elements refer to the same token. This could be used to represent the output of a POS tagger that leaves some ambiguities unresolved.

<tag> elements may also have an identifier attribute “ID”.

```
...
    </sentences>
    <POStags tagset="STTS">
      <tag tokenIDs="t1">NE</tag>
      <tag tokenIDs="t2">VVFIN</tag>
      <tag tokenIDs="t3">ART</tag>
      <tag tokenIDs="t4">NN</tag>
      <tag tokenIDs="t5">$.</tag>
      <tag tokenIDs="t6">PPER</tag>
      <tag tokenIDs="t7">VVFIN</tag>
      <tag tokenIDs="t8">PPER</tag>
      <tag tokenIDs="t9">$.</tag>
    </POStags>
  </TextCorpus>
</D-Spin>
```

6.2.5 The “lemmas” Layer

The *lemmas* layer encodes lemma information for the tokens. A <lemmas> element contains a set of individual <lemma> elements that enclose the lemma string for the token (or tokens) referred to via the “tokenIDs” attribute.

```
...
    </POStags>
    <lemmas>
      <lemma tokenIDs="t1">Peter</lemma>
      <lemma tokenIDs="t2">essen</lemma>
      <lemma tokenIDs="t3">ein</lemma>
      <lemma tokenIDs="t4">Salamipizza</lemma>
      <lemma tokenIDs="t5">.</lemma>
      <lemma tokenIDs="t6">sie</lemma>
      <lemma tokenIDs="t7">schmecken</lemma>
      <lemma tokenIDs="t8">er</lemma>
      <lemma tokenIDs="t9">.</lemma>
    </lemmas>
  </TextCorpus>
</D-Spin>
```

6.2.6 The “morphology” Layer

The *POStags* layer encodes the POS information as a flat string. The “morphology” layer provides an alternative representation for morpho-syntactic information based on *feature structures*. It is also capable of representing the internal structure of morphologically complex words.

A <morphology> element has an optional Boolean attribute “segmentation” to indicate whether word segmentation information is encoded or not and contains a set of <analysis> elements. Each <analysis> element has a “tokenIDs” attribute to refer to the annotated token(s) and contains a <tag> element and an optional <segmentation> element.

The <tag> element contains an <fs> element that encodes the morpho-syntactic information as a feature structure. The <fs> element contains a sequence of <f> elements that represent the features. Each <f> element has a *name* attribute that specifies the feature name and encloses either a character string that represents an atomic feature value or an <fs> element if the structure is recursive.

```
...
   <morphology segmentation="true">
     <analysis tokenIDs="t4">
       <tag>
         <fs>
           <f name="cat">noun</f>
           <f name="case">acc</f>
           <f name="gender">fem</f>
           <f name="number">singular</f>
         </fs>
       </tag>
       <segmentation>
         <segment cat="noun" start="0" end="10">
           <segment type="stem" cat="noun" func="comp" start="0" end="5">
             Salami
           </segment>
           <segment type="stem" cat="noun" func="base" start="6" end="10">
             Pizza
           </segment>
          </segment>
        </segmentation>
      </analysis>
    </morphology>
  </TextCorpus>
</D-Spin>
```

If the “segmentation” attribute of the <morphology> element is present and has the value “true”, then each <analysis> element also contains a <segmentation> element to encode the morphological structure of the word. The <segmentation> element contains a sequence of elements. Each element has the following optional attributes:

- “type” distinguishes between different types of morphemes such as “stem”, “prefix”, and “suffix”.
- “cat” represents the syntactic category.
- “func” further specifies the function of the morpheme such as compounding, derivation or base stem, for instance.
- “start” and “end” encode the start and end position of the morphological segment in the character string of the word form.

The element either encloses the character string if the segment consists of a single morpheme, or a sequence of elements if it is morphologically complex.

6.2.7 The “namedEntities” Layer

The *namedEntities* layer encodes the output of a named entity recognizer. The <namedEntities> element has a “tagset” attribute that specifies the set of named entity categories used, such as MUC1990, TIMEX etc. The <namedEntities> element encloses a sequence of <entity> elements. Each <entity> element has a “tokenIDs” attribute that specifies the set of tokens that the named entity consists of and a “class” attribute that specifies the class of the named entity.

```
...
   <namedEntities tagset="MUC1990">
      <entity class="PERSON" tokenIDs="t1"/>
    </namedEntities>
  </TextCorpus>
</D-Spin>
```

6.2.8 The “WordSplittings” Layer

The *WordSplittings* layer can be used to represent a syllabification or hyphenation or other kind of internal segmentation of words. The <WordSplittings> element has an attribute “type” which encodes the type of segmentation (e.g. syllabification or hyphenation) and encloses a set of <split> elements. Each <split> element encodes the segmentation of one token (which is referred to via the “tokID” attribute) by the enclosed sequence of integers that specify the positions at which the token is split.

```
...
    <WordSplittings type="hyphenation">
      <split tokID="t4">2 4 6 9</split>
    </WordSplittings>
  </TextCorpus>
</D-Spin>
```

6.2.9 The “Phonetics” Layer

The *Phonetics* layer represents the phonetic transcriptions of words. Its “transcription” attribute specifies the encoding of the phonetic symbols (e.g. IPA). The <Phonetics> element encloses a set of <pron> elements. Each <pron> element refers to a token via the “tokID” attribute and encloses a character string that represents the pronunciation of the token.

```
...
    <Phonetics transcription="IPA">
      <pron tokID="t7">....</pron>
    </Phonetics>
  </TextCorpus>
</D-Spin>
```

6.2.10 The “QueryResults” Layer

The *QueryResults* layer encodes the result of a corpus query. The “query” attribute of the <QueryResults> element has the query string itself as attribute. The <QueryResults> element encloses a set of <match> elements that represent the individual matches that the query processor has found. Each <match> element has a “tokenIDs” attribute that specifies the tokens belonging to this match result. The “tokenIDs” attribute of the optional <key> element indicates the subset of tokens that was actually matched by the query.

```
...
    <QueryResults query="[pos=VVFIN][]*[pos=NN] within s">
      <match tokenIDs="t1 t2 t3 t4">
        <key tokenIDs="t2 t4"/>
      </match>
    </QueryResults>
  </TextCorpus>
</D-Spin>
```

The <token> elements that the <match> elements refer to, must have been stored in a separate <tokens> layer. This has the advantage that those tokens can be POS tagged, lemmatized, and parsed just as any other corpus. If the query results are not complete sentences, the results of further processing steps might be suboptimal, however.

6.2.11 The “parsing” Layer

Constituent parses are represented by the *parsing* layer. The “tagset” attribute specifies the inventory of non-terminal labels used in the parse trees. Each parse tree is stored in a separate <parse> element. A <parse> element has an optional “ID” attribute and contains a single <constituent> element. Each <constituent> element has a “cat” attribute that specifies the syntactic category and an optional “ID” attribute.

<constituent> elements that correspond to pre-terminal nodes of the parse tree (i.e. POS tags) have an additional “tokenIDs” attribute that specifies a sequence of tokens (a single word or a multi-word element) that this pre-terminal category expands to. Other <constituent> elements are recursive and enclose a sequence of <constituent> elements.

```
...
  <parsing tagset="Tiger">
   <parse>
    <constituent ID="c11" cat="TOP">
     <constituent ID="c12" cat="S-TOP">
      <constituent ID="c13" cat="NP-SB">
       <constituent cat="PPER-HD-Nom" ID="c14" tokenIDs="t6"/>
      </constituent>
      <constituent cat="VVFIN-HD" ID="c15" tokenIDs="t7"/>
      <constituent ID="c16" cat="NP-DA">
       <constituent cat="PPER-HD-Dat" ID="c17" tokenIDs="t8"/>
      </constituent>
     </constituent>
     <constituent cat="\$." ID="c18" tokenIDs="t9"/>
    </constituent>
   </parse>
  </parsing>
  </TextCorpus>
</D-Spin>
```

This representation is also able to encode trees with discontinuous constituents because the tokens at the bottom layer are referenced via their IDs and their order is separately specified in the *tokens* layer.

6.2.12 The “depparsing” Layer

For dependency parses, there is a separate *depparsing* layer. The <parse> elements used here enclose a set of *dependency* relations. Each <dependency> element has a “depIDs” and a “govIDs” attribute to specify the token(s) which the dependent or governor consists of, respectively. The optional “func” attribute represents the grammatical function of the dependent.

```
...
    <depparsing>
      <parse>
        <dependency func="SUBJ" depIDs="t1" govIDs="t2"/>
        <dependency func="SPEC" depIDs="t3" govIDs="t4"/>
        <dependency func="OBJ" depIDs="t4" govIDs="t2"/>
      </parse>
      <parse>
        <dependency func="SUBJ" depIDs="t6" govIDs="t7"/>
        <dependency func="OBJ" depIDs="t8" govIDs="t7"/>
      </parse>
    </depparsing>
  </TextCorpus>
</D-Spin>
```

6.2.13 The “relations” Layer

Beyond syntactic dependencies, there are other kinds of linguistic relations between words or constituents that are encoded in corpora. Typical examples are coreference chains.

The <relations> element is a general means for the encoding of such relations. Its “type” attribute indicates the type of relation such as *coreference*. It encloses a sequence of <relation> elements. Each <relation> element has an attribute “refIDs” which specifies the sequence of entities (tokens or parse constituents etc.) that enter this relation. The optional attribute “func” provides further information on the type of the relation.

```
...
    <relations type="coreference">
      <relation refIDs="c3 c17"/>
      <relation refIDs="c7 c13"/>
    </relations>
  </TextCorpus>
</D-Spin>
```

6.2.14 The “sem_lex_rel” Layer

The *sem_lex_rel* layer represents information about WordNet relations. The “src” attribute of the <sem_lex_rel> element encodes the WordNet version from which the information was taken. Each <sem_rel> element has a “type” attribute that specifies the type of the relation (such as synonymy, hyponymy, or antonymy) and encloses a set or <orthform> elements. Each <orthform> element refers to the word whose relations are displayed via a “tokenIDs” attribute, and encloses the sequence of words that the word is related to.

```
...
    <sem_lex_rels src="GermaNet 5.3">
      <sem_rel type="synonymy"/>
        <orthform tokenIDs="t25">Orange|Apfelsine</orthform>
      </sem_rel>
    </sem_lex_rels>
  </TextCorpus>
</D-Spin>
```

6.3 The “MetaData” Element

Each TCF document must contain a <MetaData> element as a child of the <D-Spin> element. This element was not shown in the previous examples for the sake of simplicity. Currently, the only allowed element inside of <MetaData> is <source>. In the future, the <MetaData> element will be used to store metadata and provenance data for the corpus once this information becomes available. The XML schema for the <MetaData> element will then be further extended.

```
<D-Spin version="0.4">

  <MetaData>
    <source>IMS, Uni Stuttgart</source>
  </MetaData>

  <TextCorpus>
    <text>Peter aß eine Käsepizza. Sie schmeckte ihm.</text>
  </TextCorpus>
</D-Spin>
```

6.4 The “Lexicon” Element

The TCF format cannot only be used to store corpus data, but also lexical information that is encoded with a <Lexicon> element rather than a <TextCorpus> element. Thus far, the <Lexicon> element was primarily used to encode the result of a collocation extraction web service.

The <Lexicon> element has an optional attribute “lang” which specifies the language. Possible child elements (= annotation layers) of the <Lexicon> element are *lemmas*, *POStags*, *frequencies*, and *word-relations*.

The <lemmas> element specifies the basic entities of a Lexicon document. It contains a set of <lemma> elements. Each <lemma> element encloses the character string of a lemma and the optional attribute “ID” contains a unique identifier as value.

The <POStags> element encodes part-of-speech information for the lemmas. Its “tagset” attribute specifies the tag inventory from which the POS tags are taken. The <POStags> element contains a set of <tag> elements. Each <tag> element encloses a POS tag string for a lemma that is referred to via a “lemID” attribute.

In a similar way, the <frequencies> element contains a set of <frequency> child elements each of which specifies the frequency of a lemma. The lemma is again referred to via a “lemID” attribute and the frequency is represented as an integer value that is enclosed by the <frequency> element.

Finally the <word-relations> element is used to encode collocations and other word-word relations. <word-relation> encloses a set of <word-relation> elements. Each <word-relation> element has optional attributes “type”, “func”, and “freq” to encode the type of the relation (e.g. “syntactic relation”), a further functional specification (e.g. “verb+direct-object”) and the frequency of occurrence, respectively. A <word-relation> element contains a sequence of at least 2 <term> elements that specify the terms that enter the word relation, and any number of <sig> elements. Each <term> element specifies a lemma either directly by enclosing its character string or by referencing it via a “lemID” attribute. The <sig> elements are used to encode the results of different word association measures. The “measure” attribute indicates the word association measure used such as “mutual information” or “log-likelihood ratio”.

The following example shows a Lexicon document complete with MetaData and name-space information:

```
<D-Spin xmlns="http://www.dspin.de/data" version="0.4">

  <MetaData xmlns="http://www.dspin.de/data/metadata">
    <source>IMS, Uni Stuttgart</source>
  </MetaData>

  <Lexicon xmlns="http://www.dspin.de/data/lexicon" lang="de">
    <lemmas>
      <lemma ID="l1">halten</lemma>
      <lemma ID="l2">Vortrag</lemma>
    </lemmas>
    <POStags tagset="basic">
      <tag lemID="l1">Verb</tag>
      <tag lemID="l2">Noun</tag>
    </POStags>
    <frequencies>
      <frequency lemID="l1">1257</frequency>
      <frequency lemID="l2">193</frequency>
    </frequencies>
```

```
    <word-relations>
      <word-relation type="syntactic relation" func="verb+direct-object" freq="13">
        <term lemID="l1"/><term lemID="l2"/>
        <sig measure="MI">13.58</sig>
      </word-relation>
    </word-relations>
  </Lexicon>
</D-Spin>
```

7 Description of Visualization

Visualization of linguistic data is important for its analysis. While xml is text-based and hence human readable to some degree, it is not intended to be read by humans. The linguistic annotations contained in a TCF file are not easily conceived by simply inspecting the xml. Annotation Viewer is a visualization tool developed as a part of WebLicht and in accordance with SOA (Service Oriented Architecture) principles. It presents linguistic annotations using common graphical structures such as tables and trees

WebLicht web application integrates Annotation Viewer by offering the user an option to view tool chain processing results in an integrated graphical interface (See Figure 8).

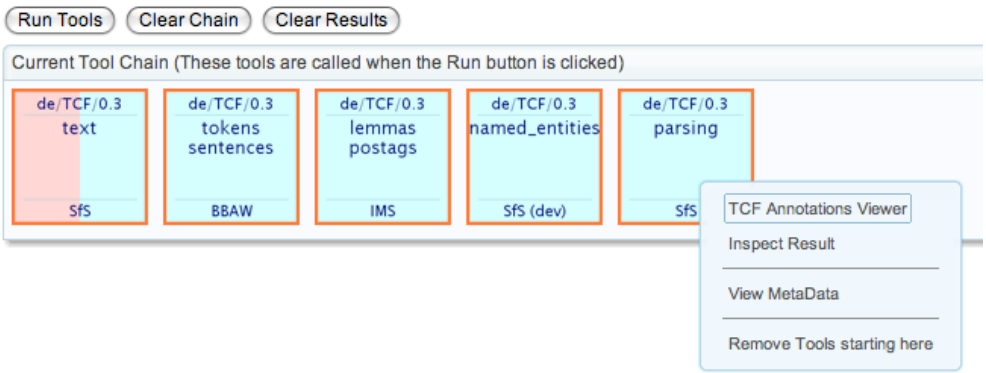


Figure 8: WebLicht user interface

The visualization tool can also be used as a stand-alone web application available over the net (available at http://weblicht.sfs.uni-tuebingen.de:8080/visual/index.zul). It can either be called from another application or used directly. In the latter case the users can upload data in TCF0.3 format and get a graphical view of their linguistic annotations.

Currently Annotation Viewer visualizes data present in lexicon and text corpus format TCF0.3, such as text, sentence annotations, tokens, lemmas, part-of-speech tags, named entity tags, lexical-semantic relations and parsing annotations. Inline annotations are visualized in a table and users can select to view the annotation type or types they are interested in (See Figure 9).

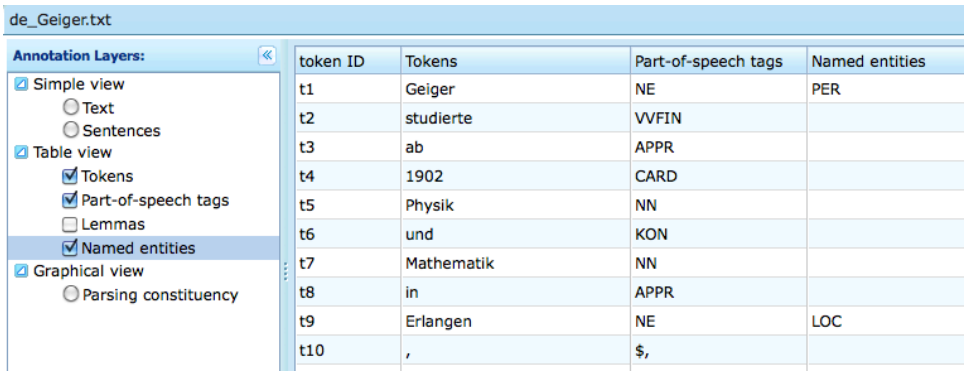


Figure 9: Annotation Viewer

Parsing annotations are visualized as tree images (see Figure 10).

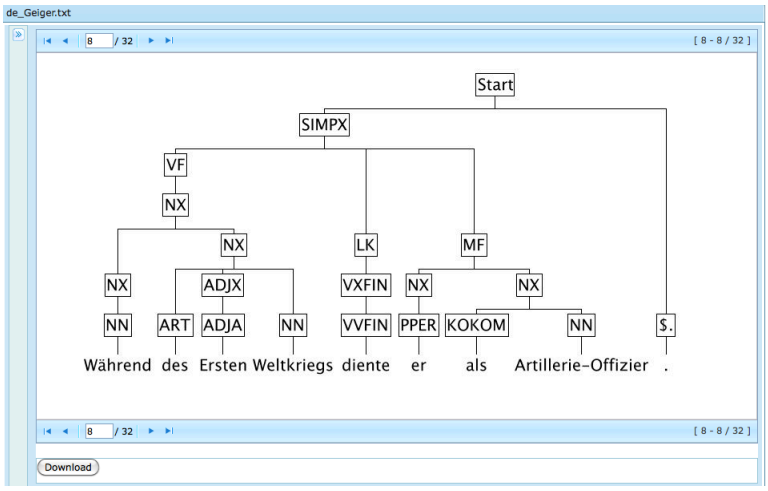


Figure 10: Tree images

The tree images are scrollable if their size exceeds the view area (see Figure 11).

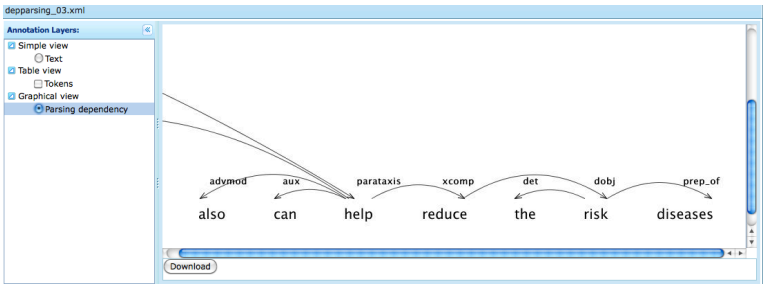


Figure 11: Scrollable tree image

The images can be downloaded in jpg, pdf and svg formats.

Under the hood Annotation Viewer implementation uses a StAX-based library to read the data, and ZK Ajax framework to visualize the data. StAX parser allows for efficient use of CPU and RAM. The Java library, built on top of the StAX parser to bind the text corpus format allows for easy data access. The ZK toolkit provides a rich user experience. The benefits of the visualization tool for the end users is that they do not need to know anything about the internal implementation, they do not

need to install anything and can run it on a computer with any operating system which has a modern browser available.

In future work, the Annotation Viewer will be made more interactive, to allow user not only to view, but also to edit linguistic annotations.

8 General Description of Tools

As mentioned in section 3.5, WebLicht offers a variety of web services, including tokenizers, sentence border detection, part-of-speech taggers, named entity recognition, lemmatization, constituent parsing, cooccurrence annotation, a semantic annotator, and data format converters. The purpose of this section is to give a general idea of what these tools do.

8.1 Tokenizer

Tokenizing is the task of segmenting raw text into *tokens*, basic lexical units, numbers, and punctuation. A tokenizer is a computer program for this task.

Tokenizing might seem a trivial task since often, the white space character acts as boundary marker. However, the white space character can also occur within a token (for example in Multi-Word Expressions, such as *in spite of*) or not occur between tokens (for example, punctuation is generally adjacent to its preceding or succeeding word). In East Asian languages, tokenization is particularly difficult since by default, in these languages, whitespace acts as syllable separator.

For each language for which there are services available in WebLicht, there is at least one tokenizer.

8.2 Sentence Boundary Detection

Sentence boundary detection is the task of marking sentence boundaries in tokenized text. A sentence boundary detector is a computer program for this task.

A sentence boundary detector faces challenges that are similar to the challenges a tokenizer faces. In many cases, punctuation can give an orientation. However, in other cases, it can be misleading, for example in the case of embedded direct speech. Additionally, the use of punctuation might vary over different text types.

For each language for which there are services available in WebLicht, there is at least one sentence boundary detector.

8.3 Part-of-Speech Tagger

Part-of-speech tagging (POS tagging) is the task of marking tokens in a text as having a particular part-of-speech. A POS tagger is a computer program for this task.

In order to tag single tokens, a text must be tokenized. While older systems were hand-constructed rule-based systems, newer systems work on the basis of statistical models. They typically achieve very high accuracy rates of well over 90%. Popular statistical taggers are MXPOST (Ratnaparkhi, 1996), T'n'T (Brants, 2000) and the TreeTagger (Schmid, 1993). Over the years, in the NLP/LRT community, certain POS tag sets have been established as a de facto standard. For example, for English, this is the set of POS tags used in the Penn Treebank. For German, it is the Stuttgart-

Tübingen Tagset (STTS) (Schiller, Teufel, & Thielen, 1995) which is used in all major German treebanks.

WebLicht provides POS taggers services for different languages. For German, e.g., it provides the TreeTagger, which can POS tag texts using the STTS.

8.4 Named Entity Recognizer

Named entity recognition (NER) is the task of marking those elements in a text that correspond to categories such as the names of persons, locations, or organizations. A NE recognizer is a computer program for this task. The NER task got particular attention in the mid-90s with the Message Understanding Conferences (Grishman & Sundheim, 1995). More recent systems use sophisticated statistical models and achieve near-human performance (see, e.g., (Finkel, Grenager, & Manning, 2005)).

WebLicht offers, amongst others, the MaxEnt-based Named Entity Recognizer belonging to OpenNLP (see http://incubator.apache.org/opennlp/).

8.5 Lemmatizer

Lemmatization is the task of determining the lemma for a given word. A *lemmatizer* is a computer program for this task. In order to determine the lemma of the word, knowledge about the context of the word is necessary, as well as knowledge about its part-of-speech.

WebLicht offers lemmatizer services for different languages.

8.6 Parsing

Parsing is the process of automatic grammatical analysis of text. A *parser* takes as input a bare (tokenized) sentence and annotates it with syntactic information.

A *constituent* is a sequence of words that form a linguistically meaningful unit. It is labeled with its linguistic type, e.g., noun phrase (NP), verb phrase (VP), etc. A hierarchical structure of constituents is called a *constituency structure*. A *constituency parser* aims at constructing a constituency structure over a bare tokenized sentence. Rules for building a constituency structure are generally formulated as a *grammar*. Such a grammar can be written by hand. In *probabilistic parsers*, however, the grammar is obtained from a *treebank,* a collection of sentences that have been hand-annotated by linguists with constituency structures. These constituency structures are then assumed to be the product of a latent grammar and an appropriate algorithm is used to extract the grammar rules. For each rule, occurrence probabilities are computed. With such a probabilistic grammar, a probabilistic parser can build the most probable structure for a sentence and deliver it to the user.

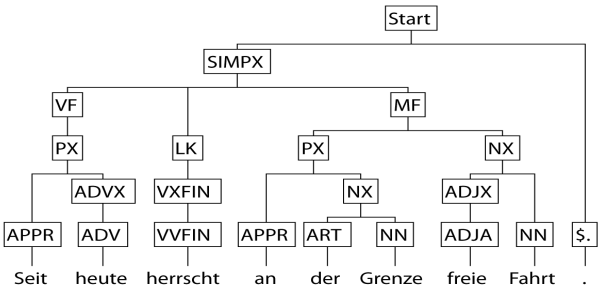


Figure 12: Berkeley Parser output, exported from the WebLicht Annotations Viewer

Consider Figure 12 as an example of how the output of a constituency parser looks like.

For an overview on probabilistic parsing and many pointers to further literature, consult (Cahill, 2008). At the moment, WebLicht offers two constituency parser services, LoPar (http://www.ims.uni-stuttgart.de/tcl/SOFTWARE/LoPar.html) (Schmid, 2000) and the Berkeley Parser (http://code.google.com/p/berkeleyparser/) (Petrov, Barrett, Thibaux, & Klein, 2006).

8.7 Cooccurrence Counting

A *cooccurrence* is a joint appearance of two tokens within a certain distance in a text. The window in which the two tokens occur can vary in length (e.g., left/right neighbors vs. occurrence in the same sentence) (Church & Hanks, 1990). A cooccurrence counter is a computer program that counts cooccurrences and computes statistics on them.

WebLicht offers a cooccurrence counting service for German based on the Wortschatz portal of the University of Leipzig (http://wortschatz.uni-leipzig.de).

8.8 Semantic Annotation

A word net, generally identified with its most popular incarnation, the Princeton WordNet (Fellbaum, 1998), describes lexical meanings. Unlike a traditional lexicon, it is organized directly around the meanings. WordNet allows identifying semantic relations between different words, such as hypernymy/hyponymy, meronymy/holonymy, etc.

WebLicht offers a service that uses GermaNet (Hamp & Feldweg, 1997), the German word net, for the semantic annotation of lexical items in the processed German text.

8.9 Data Format Conversion

While the WebLicht services internally rely on the TCF data format (see section 6), there is a landscape of other formats that may serve as input formats or be desired as output formats. The output formats can thereby also be formats which represent a partial aspect of the data. WebLicht offers converters for MS Word/PDF/RTF to TCF, plain text to TCF, TCF to/from MAF, Negra and TEI, TCF to KML (Geo mapping of locations to be viewed, e.g., with Google Earth).

9 Project Partners

9.1 C4 corpus

9.1.1 Composition of the Corpus

The C4-Korpus is a common effort of the Berlin-Brandenburg Academy of Sciences, the Austrian Academy of Sciences, the University of Basel (Schweizer Textkorpus) and the Free University of Bozen (Korpus Südtirol). The history of the C4-Korpus reaches back to the year 2006, when the four partners formally agreed to construct a shared corpus. The common goal was to provide a reference corpus for the study of regional varieties of standard German. Each partner is supposed to supply 20 million tokens from their corpus materials. The C4 corpus is not yet completed. The corpus currently consists of 20 Million tokens of the German and Swiss Corpus each, 4.1 million tokens from the Austrian corpus and 1.7 tokens from the corpus of South Tyrol. However, it can already be queried

e.g. through the DWDS-website of the Berlin-Brandenburg Academy of Sciences (www.dwds.de) and by a portal which is offered by the University of Basel http://chtk.unibas.ch/korpus-c4/search).

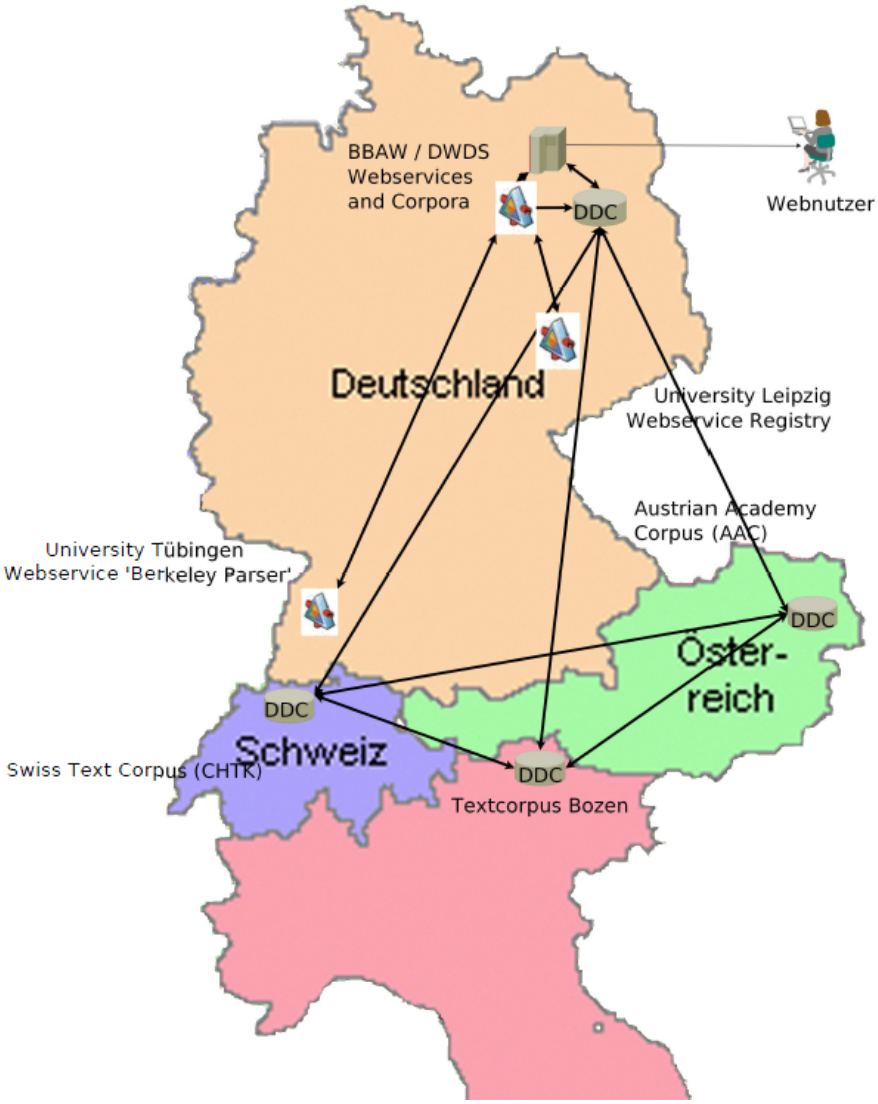


Figure 13: Distribution of the C4-Korpus and related services

This collection of texts is virtual in a sense that there is no place (server etc.) where all the texts reside. Indeed the four parts of this corpus reside on the servers of the donating institutions. Nevertheless they can be considered, and, even more important, queried as a whole (see Figure 13). The heart of this corpus is the search engine DDC ("DWDS Dialing Concordance"). This search engine has been implemented by Alexey Sokirko (Sokirko, 2003) for the DWDS project and has been provided since then as a web service by the DWDS team.

The interaction and data flow is presented in Figure 14.

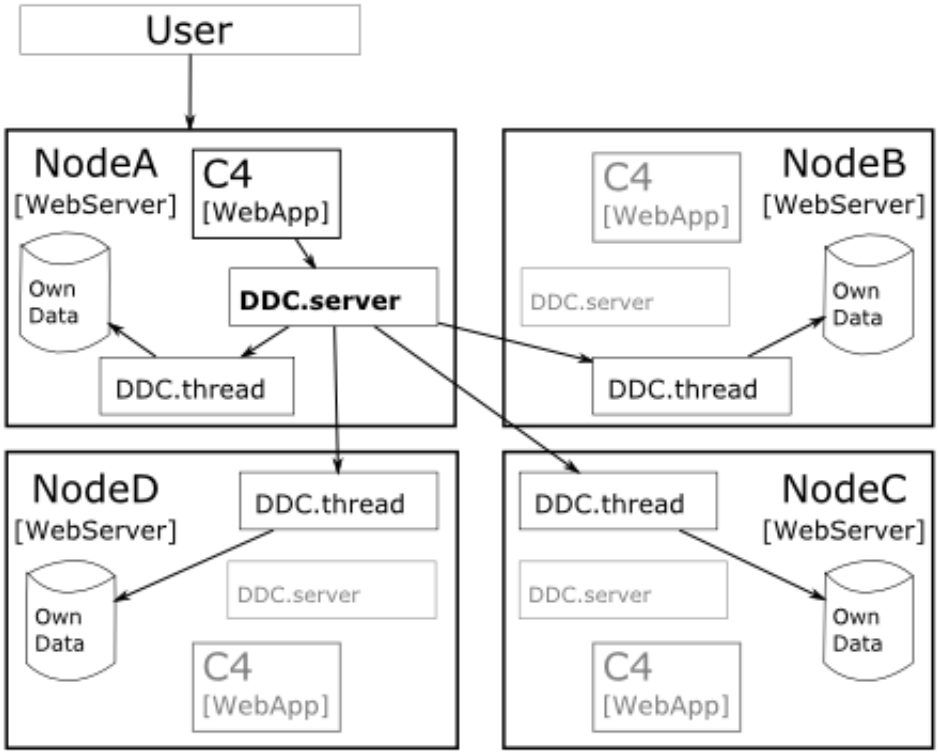


Figure 14: Interaction and data flow

9.1.2 The CLARIN/D-SPIN Demonstrator

With a CLARIN demonstrator case – a collaborative effort between the Berlin-Brandenburg Academy of Sciences, DWDS group, the Austrian Academy of Sciences, Institut für Corpuslinguistik und Texttechnologie (ICLTT), the University of Basel (Schweizer Textkorpus) and the Free University of Bozen (Korpus Südtirol) – we want to show how simple it is to integrate WebLicht web services into external applications.

Currently, WebLicht provides more than 90 tools and resources, among them tools for basic text-technological operations, e.g. tokenizing, part-of speech tagging, parsing, and semantic annotation. All tools use a standardized XML-based exchange format named TCF.

These tools can be plugged together into larger workflow chains.

As a preparation for the demonstrator, DWDS added a new C4 corpus web service to the WebLicht tools collection, so that the corpus can be used by all WebLicht users and in all possible tool chain combinations.

The starting point of the presented demo is a query on the C4-corpus, launched from the DWDS website. As a result a set of concordance lines is retrieved for this search expression and shown to the user. Each concordance line is embellished with an extra button for triggering further linguistic analyses (cf. Figure 15).

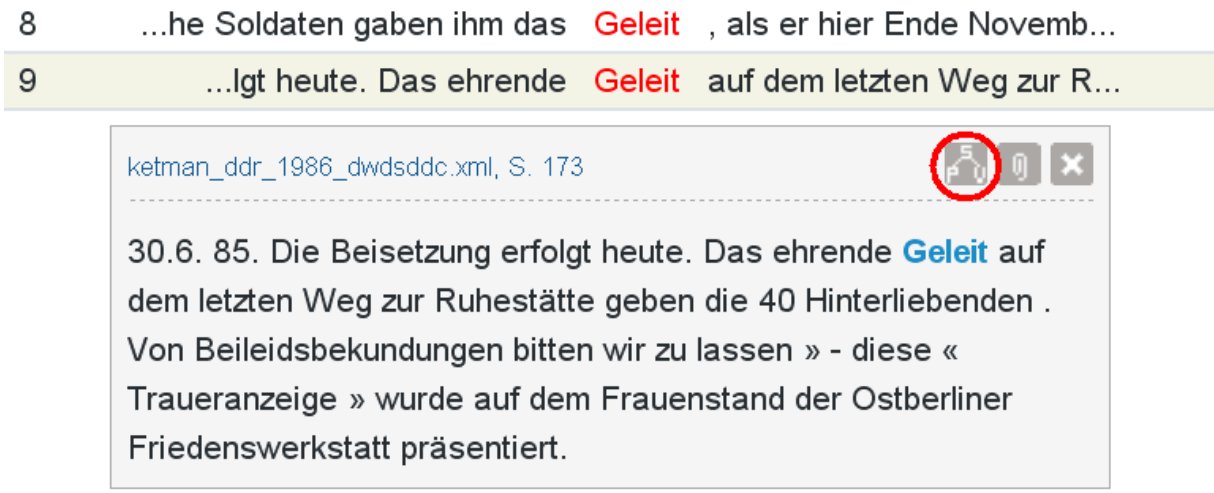


Figure 15: C4-Concorandance line with the extra function "linguistic analysis"

Clicking this button triggers a pre-configured tool chain in WebLicht that produces a linguistically enriched version of the input sentence that has been sent for processing. Last step in the workflow is the visualization of the sentence. For the demonstrator, we use standard tools (SVG graph is generated by XML transformation, see Figure 16).

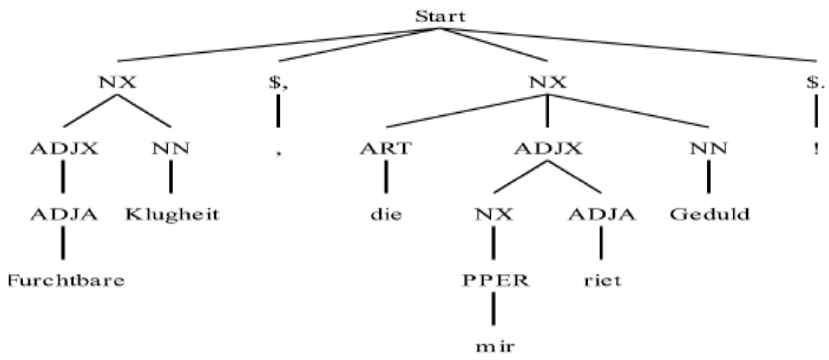


Figure 16: Final output of a linguistically analyzed example sentence from the C4 corpus

To sum up: based on the results of a query on our distributed corpus, the demonstrator produces a linguistically enriched version of a result sentence and presents this sentence visually to the user.

9.1.3 Further work

First, means have to be provided for the user to store the resulting data structures, i.e. the analyzed and annotated sentences. The user might want to further process these results. Therefore, a workspace is needed.

Second, we are now working with a pre-configured workflow that is triggered by the “tree view” button. Future users might want to build their own processing and visualization chains. Means have to be provided to support this functionality.

9.2 International Tools

External international institutions and individuals have supported the project by providing tools in the form of web services, including:

- Department of Modern Languages, University of Helsinki, Finland
    - Tokenizer for Finnish
    - Morphological Analyzer for Finnish
    - Plain-text to TCF converter for Finnish
- Romanian Academy Research Institute for Artificial Intelligence (RACAI), Bucharest, Romania
    - Chunkers for English, French and Romanian
    - Lemmatizers for English, French and Romanian
    - Tokenizer/POS Taggers for English, French and Romanian
    - Plain-text to TCF converters for English, French and Romanian
- Institute of Computer Science, Polish Academy of Sciences (IPIPAN), Warsaw, Poland
    - Plain-text to TEI for Polish, which produces paragraphs, POS tags, lemmas, tokens, and morphological analysis, and sentence boundaries
    - Shallow parser for Polish, including syntactic words
    - Deep parser for Polish

Currently, no further integration of further web services is planned.

10 Tutorials

10.1 Getting Started with WebLicht

To use WebLicht to build and execute linguistic tools, the following steps must be taken:

- Specify the input text and parameters
- Build a tool chain
- Run the tool chain on the input text and view the results

10.1.1 Input

Input text can be entered several ways:

1. Enter plain text:

- Choose "Enter Plain Text" in the **Input Options**
- Edit the text in the **Text Input** area

2. Use a sample file:

- Choose "Use Sample File" in the **Input Options**
- Select a sample text from the list
- The contents of the file will be displayed in the **Text Input** area.

3. Upload a file from your computer:

- Choose "Use Uploaded File" in the **Input Options**
- If you have already uploaded one or more files you can choose one to use
- Click the "Upload File..." button to select a file from your computer.

It is possible to upload any type of file for which services are available. Plain text files can be processed in the same way as plain text entered in the **Text Input** area. TCF files can be uploaded

and further processed. Currently it is also possible to upload doc, rtf, TEI, and MAF files, which are generally converted to TCF in the first step of the processing chain. However, this conversion is not a requirement of the infrastructure. If services are available that operate directly on a non-TCF format, then those services will be offered when building the processing chain.

Uploaded files are used only for the purpose of providing input text, are not accessible to other users, and are deleted when your session ends. Figure 17 shows the WebLicht input dialog.

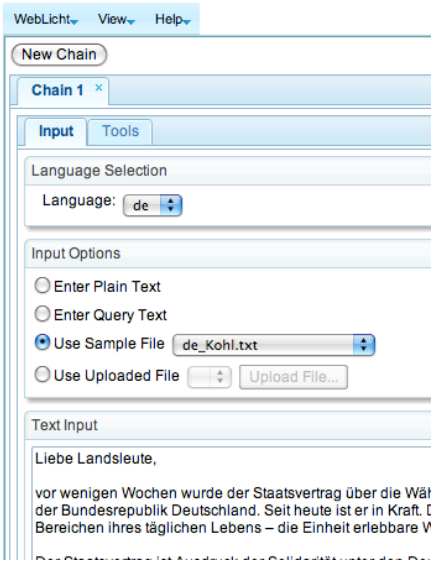


Figure 17: WebLicht Input

10.1.2 Building a Tool Chain

After setting the input parameters, move to the **Tools** tab. Drag a tool icon from the **Next Choices** area to the **Chosen Tools** area. The icons in the **Next Choices** area will then be automatically updated so that they are compatible with the tools that have already been added to the chain. Continue this process to add additional tools.

Figure 18 shows the corresponding WebLicht dialog.

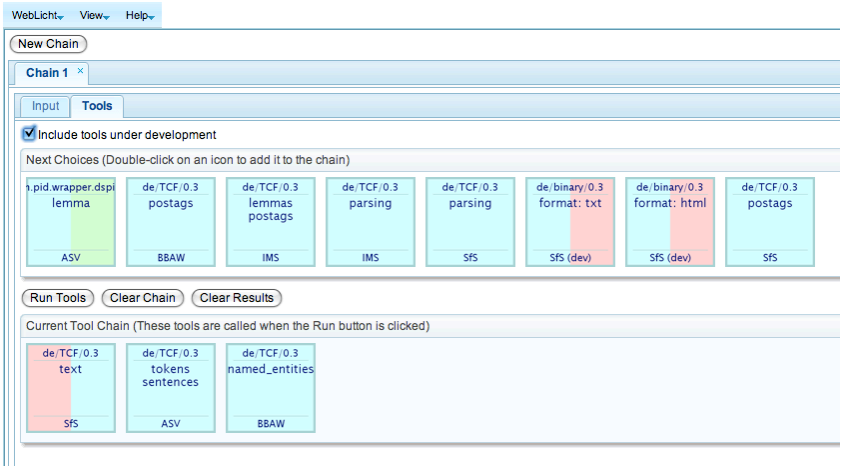


Figure 18: Building a tool chain

10.1.3 Running a Tool Chain

You can run the selected tools by clicking the **Run Tools** button. When the processing is finished, the tool icons in the **Next Choices** area will change background color. Tools can still be added to the chain after running. In this case, when the **Run Tools** button is pressed, only those tools that have not yet been run will be invoked. Status messages are given in the status bar of the browser. The running chain can be interrupted by clicking the corresponding button.

10.1.4 View Results

Results can be viewed by right clicking on a tool icon in the **Chosen Tools** area and selecting a viewer. Currently, there are two choices for viewing results:

- **TCF Annotations Viewer** by Yana Panchenko (SfS Tübingen) (see section 7) offers a visual view of the annotation layers contained in the result data. Results are displayed in a new browser tab. Download of parse images are available.
- **Inspect Result** shows the raw XML result in a sub-window within WebLicht. Download of the xml is available.

10.1.5 Create a New Chain

Click the **New Chain** button. A new chain tab will open. Select input and tools as described above.

10.2 Tutorial for Creating WebLicht Web Services in Java

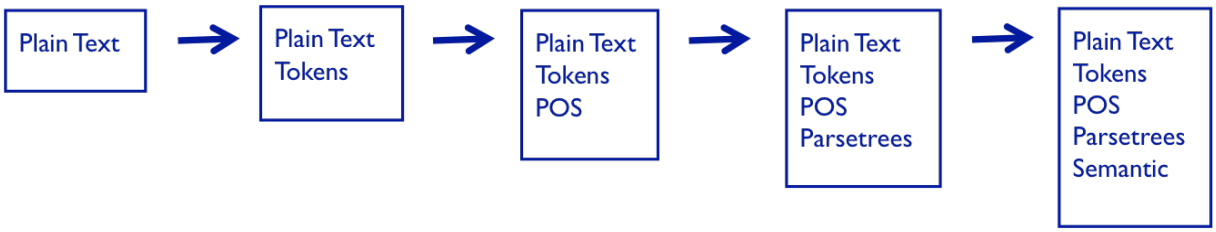
This tutorial describes the implementation of WebLicht web services in Java. Familiarity with the Java programming language, XML documents, and WebLicht is assumed. This tutorial can be downloaded from http://weblicht.sfs.uni-tuebingen.de/weblichttutorial.shtml

10.2.1 WebLicht Overview

Web services for the WebLicht tool chain are implemented as RESTful web services. The input is sent via the POST method of the HTTP protocol to the service. The output of the web service is the response to that POST event.

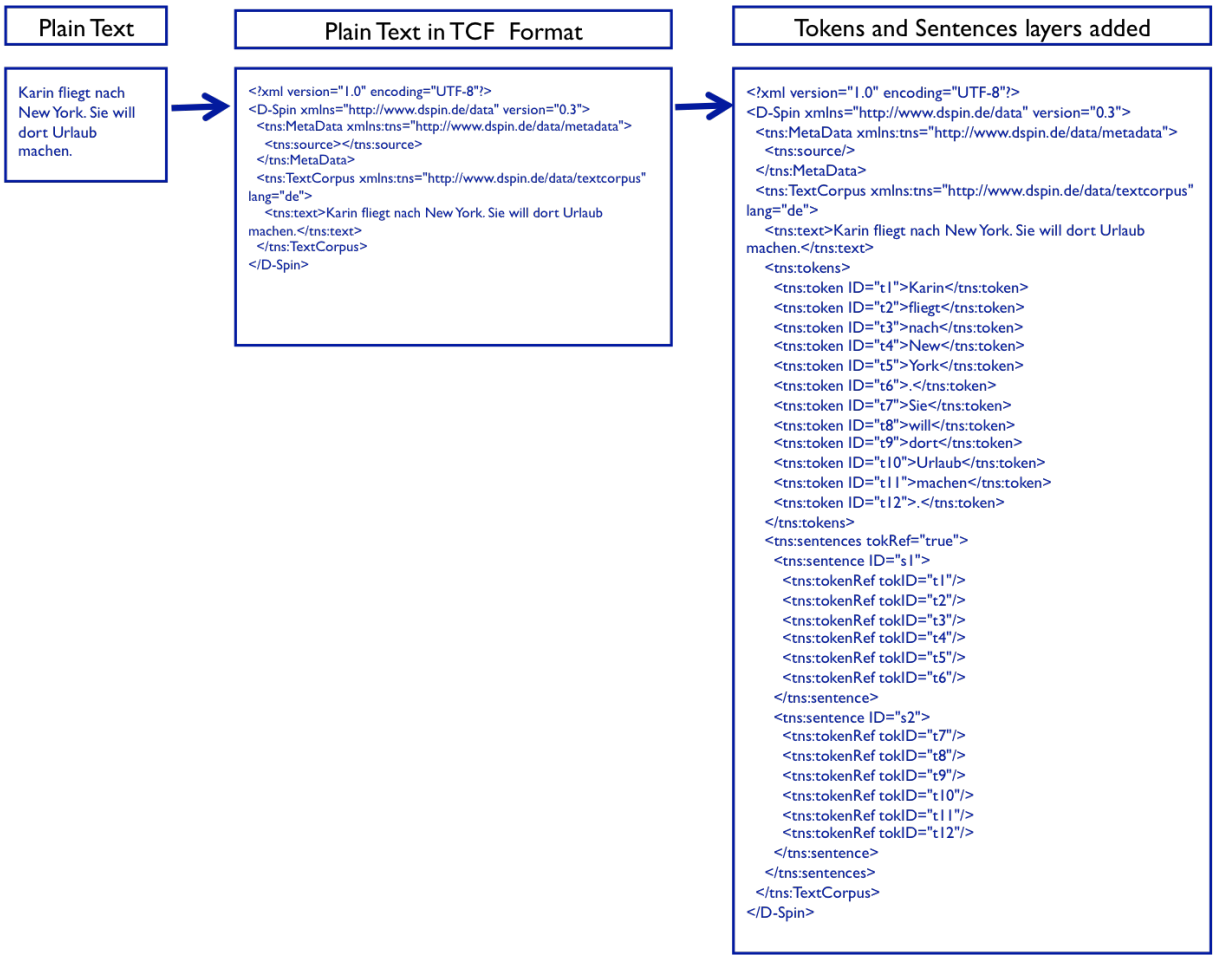
10.2.2 The TCF Format

The TCF format (see section 6) is a simple standoff format for linguistic annotated text corpora. In contrast to some other formats (for example MAF), TCF stores all linguistic layers in one file. That means that during the chaining process, the file grows. A typical chain might look like this:



WebLicht is not restricted to the use of the TCF format, although most of the services currently registered do use it. The advantage is that services using the same format can be easily chained together. This tutorial assumes use of the TCF format.

Important: every web service can add an arbitrary number of layers to the TCF file, but no web service should change or remove an already existing layer! This can be seen in the following sample input and output of a typical chain, where the output of one service is used as input to the next.



10.2.3 Example Service

The sample service included in this tutorial analyzes the occurrences of lemmas and tokens in the input document. A TCF layer called “uniqueLemmas” is created, which contains one entry per lemma, each entry in turn containing the corresponding tokens and the number of times which that token appears.

In order to proceed with the sample service included in this tutorial, you will need:

- A Java servlet container, such as Tomcat, Glassfish, or JBoss.
- An IDE, such as NetBeans or Eclipse. The project was developed with NetBeans, but it is possible to import it into Eclipse.
- A command-line tool such as `wget` or `curl` for testing

At http://weblicht.sfs.uni-tuebingen.de/WebLichtTutorial.zip, you can download the example files necessary for this tutorial.

- WebLichtSampleService.war: can be deployed to a Java servlet container
- sampleInput.xml: sample input for the service
- WebLichtSampleService.zip: unzip and open in IDE to view source code and build WAR file
- dspin_0_3.xsd, dspin_textcorpus_0_3.xsd: TCF schemas

Now we can take a closer look at the example. The sample service accepts as input a TCF document containing tokens and lemmas (possibly other layers also). From the tokens and lemmas layers, we will produce a map with the unique lemmas as keys and a list of respective tokens as values. The input document will be parsed with a StAX parser, where the tokens and lemmas will be extracted. During parsing, the document will be written to the response stream to preserve the original document. Before writing the TextCorpus close tag, we will insert our new layer. Finally, the remaining portion of the document will be written to the output as-is.

See the following pages for an example input to our service and the generated output. Notice that we only add information, but do not change or delete anything in the input.

Note that this service is not necessarily a good candidate for inclusion in WebLicht, since it only summarizes information already in the input. However, it will suffice as an example of how to create a service.

This sample service was developed with Java using the NetBeans IDE, and can be deployed in any Java servlet container, such as Tomcat, Glassfish, or JBoss.

Sample Input for the LemmaSummarizer Service:

```
<?xml version="1.0" encoding="UTF-8"?>
<D-Spin xmlns="http://www.dspin.de/data" version="0.3">
  <tns:MetaData xmlns:tns="http://www.dspin.de/data/metadata">
    <tns:source/>
  </tns:MetaData>
  <tns:TextCorpus xmlns:tns="http://www.dspin.de/data/textcorpus" lang="de">
    <tns:text>Karin aß Pizza. Max hat Pizza gegessen. Sie essen Pizza.</tns:text>
    <tns:tokens>
      <tns:token ID="t1">Karin</tns:token>
      <tns:token ID="t2">aß</tns:token>
      <tns:token ID="t3">Pizza</tns:token>
      <tns:token ID="t4">.</tns:token>
      <tns:token ID="t5">Max</tns:token>
      <tns:token ID="t6">hat</tns:token>
      <tns:token ID="t7">Pizza</tns:token>
      <tns:token ID="t8">gegessen</tns:token>
      <tns:token ID="t9">.</tns:token>
      <tns:token ID="t10">Sie</tns:token>
      <tns:token ID="t11">essen</tns:token>
      <tns:token ID="t12">Pizza</tns:token>
      <tns:token ID="t13">.</tns:token>
    </tns:tokens>
    <tns:sentences tokRef="true">
      <tns:sentence ID="s1">
        <tns:tokenRef tokID="t1"/>
        <tns:tokenRef tokID="t2"/>
        <tns:tokenRef tokID="t3"/>
        <tns:tokenRef tokID="t4"/>
      </tns:sentence>
      <tns:sentence ID="s2">
        <tns:tokenRef tokID="t5"/>
        <tns:tokenRef tokID="t6"/>
        <tns:tokenRef tokID="t7"/>
        <tns:tokenRef tokID="t8"/>
        <tns:tokenRef tokID="t9"/>
      </tns:sentence>
      <tns:sentence ID="s3">
        <tns:tokenRef tokID="t10"/>
        <tns:tokenRef tokID="t11"/>
        <tns:tokenRef tokID="t12"/>
        <tns:tokenRef tokID="t13"/>
      </tns:sentence>
    </tns:sentences>
    <tns:POStags tagset="STTS">
      <tns:tag tokID="t1">NE</tns:tag>
      <tns:tag tokID="t2">VVFIN</tns:tag>
```

```
      <tns:tag tokID="t3">NN</tns:tag>
      <tns:tag tokID="t4">$.</tns:tag>
      <tns:tag tokID="t5">NE</tns:tag>
      <tns:tag tokID="t6">VAFIN</tns:tag>
      <tns:tag tokID="t7">NN</tns:tag>
      <tns:tag tokID="t8">VVPP</tns:tag>
      <tns:tag tokID="t9">$.</tns:tag>
      <tns:tag tokID="t10">PPER</tns:tag>
      <tns:tag tokID="t11">VVFIN</tns:tag>
      <tns:tag tokID="t12">NN</tns:tag>
      <tns:tag tokID="t13">$.</tns:tag>
    </tns:POStags>
    <tns:lemmas>
      <tns:lemma tokID="t1">Karin</tns:lemma>
      <tns:lemma tokID="t2">essen</tns:lemma>
      <tns:lemma tokID="t3">Pizza</tns:lemma>
      <tns:lemma tokID="t4">.</tns:lemma>
      <tns:lemma tokID="t5">Max</tns:lemma>
      <tns:lemma tokID="t6">haben</tns:lemma>
      <tns:lemma tokID="t7">Pizza</tns:lemma>
      <tns:lemma tokID="t8">essen</tns:lemma>
      <tns:lemma tokID="t9">.</tns:lemma>
      <tns:lemma tokID="t10">Sie|sie|sie</tns:lemma>
      <tns:lemma tokID="t11">essen</tns:lemma>
      <tns:lemma tokID="t12">Pizza</tns:lemma>
      <tns:lemma tokID="t13">.</tns:lemma>
    </tns:lemmas>
  </tns:TextCorpus>
</D-Spin>
```


Sample Output of the LemmaSummarizer Service:


```
<?xml version="1.0" encoding="UTF-8"?>
<D-Spin version="0.3" xmlns="http://www.dspin.de/data">
  <tns:MetaData xmlns:tns="http://www.dspin.de/data/metadata">
    <tns:source></tns:source>
  </tns:MetaData>
  <tns:TextCorpus lang="de" xmlns:tns="http://www.dspin.de/data/textcorpus">
    <tns:text>Karin aß Pizza. Max hat Pizza gegessen. Sie essen Pizza.</tns:text>
    <tns:tokens>
      <tns:token ID="t1">Karin</tns:token>
      <tns:token ID="t2">aß</tns:token>
      <tns:token ID="t3">Pizza</tns:token>
      <tns:token ID="t4">.</tns:token>
      <tns:token ID="t5">Max</tns:token>
      <tns:token ID="t6">hat</tns:token>
      <tns:token ID="t7">Pizza</tns:token>
      <tns:token ID="t8">gegessen</tns:token>
      <tns:token ID="t9">.</tns:token>
      <tns:token ID="t10">Sie</tns:token>
      <tns:token ID="t11">essen</tns:token>
      <tns:token ID="t12">Pizza</tns:token>
      <tns:token ID="t13">.</tns:token>
    </tns:tokens>
    <tns:sentences tokRef="true">
      <tns:sentence ID="s1">
        <tns:tokenRef tokID="t1"></tns:tokenRef>
        <tns:tokenRef tokID="t2"></tns:tokenRef>
        <tns:tokenRef tokID="t3"></tns:tokenRef>
        <tns:tokenRef tokID="t4"></tns:tokenRef>
      </tns:sentence>
      <tns:sentence ID="s2">
        <tns:tokenRef tokID="t5"></tns:tokenRef>
        <tns:tokenRef tokID="t6"></tns:tokenRef>
        <tns:tokenRef tokID="t7"></tns:tokenRef>
        <tns:tokenRef tokID="t8"></tns:tokenRef>
        <tns:tokenRef tokID="t9"></tns:tokenRef>
      </tns:sentence>
      <tns:sentence ID="s3">
        <tns:tokenRef tokID="t10"></tns:tokenRef>
        <tns:tokenRef tokID="t11"></tns:tokenRef>
        <tns:tokenRef tokID="t12"></tns:tokenRef>
        <tns:tokenRef tokID="t13"></tns:tokenRef>
      </tns:sentence>
    </tns:sentences>
    <tns:POStags tagset="STTS">
      <tns:tag tokID="t1">NE</tns:tag>
      <tns:tag tokID="t2">VVFIN</tns:tag>
      <tns:tag tokID="t3">NN</tns:tag>
      <tns:tag tokID="t4">$.</tns:tag>
      <tns:tag tokID="t5">NE</tns:tag>
      <tns:tag tokID="t6">VAFIN</tns:tag>
      <tns:tag tokID="t7">NN</tns:tag>
      <tns:tag tokID="t8">VVPP</tns:tag>
      <tns:tag tokID="t9">$.</tns:tag>
      <tns:tag tokID="t10">PPER</tns:tag>
      <tns:tag tokID="t11">VVFIN</tns:tag>
      <tns:tag tokID="t12">NN</tns:tag>
```

```
      <tns:tag tokID="t13">$.</tns:tag>
    </tns:POStags>
    <tns:lemmas>
      <tns:lemma tokID="t1">Karin</tns:lemma>
      <tns:lemma tokID="t2">essen</tns:lemma>
      <tns:lemma tokID="t3">Pizza</tns:lemma>
      <tns:lemma tokID="t4">.</tns:lemma>
      <tns:lemma tokID="t5">Max</tns:lemma>
      <tns:lemma tokID="t6">haben</tns:lemma>
      <tns:lemma tokID="t7">Pizza</tns:lemma>
      <tns:lemma tokID="t8">essen</tns:lemma>
      <tns:lemma tokID="t9">.</tns:lemma>
      <tns:lemma tokID="t10">Sie|sie|sie</tns:lemma>
      <tns:lemma tokID="t11">essen</tns:lemma>
      <tns:lemma tokID="t12">Pizza</tns:lemma>
      <tns:lemma tokID="t13">.</tns:lemma>
    </tns:lemmas>
    <tns:uniqueLemmas>
        <tns:uniqueLemma lemma="essen">
            <tns:count token="aß">1</tns:count>
            <tns:count token="essen">1</tns:count>
            <tns:count token="gegessen">1</tns:count>
        </tns:uniqueLemma>
        <tns:uniqueLemma lemma="Karin">
            <tns:count token="Karin">1</tns:count>
        </tns:uniqueLemma>
        <tns:uniqueLemma lemma="Max">
            <tns:count token="Max">1</tns:count>
        </tns:uniqueLemma>
        <tns:uniqueLemma lemma="Sie|sie|sie">
            <tns:count token="Sie">1</tns:count>
        </tns:uniqueLemma>
        <tns:uniqueLemma lemma=".">
            <tns:count token=".">3</tns:count>
        </tns:uniqueLemma>
        <tns:uniqueLemma lemma="haben">
            <tns:count token="hat">1</tns:count>
        </tns:uniqueLemma>
        <tns:uniqueLemma lemma="Pizza">
            <tns:count token="Pizza">3</tns:count>
        </tns:uniqueLemma>
    </tns:uniqueLemmas>
</tns:TextCorpus>
</D-Spin>
```


10.2.3.1 WAR-files, Servlets, and URLs

Deploy WebLichtSampleService.war and go to: http://localhost:8080/WebLichtSampleService in your browser, replacing localhost:8080 with the actual host:port if necessary.

You should end up at the index.jsp page, which is entered as a welcome-file in WEB-INF/web.xml. No welcome page is needed for a WebLicht service because it is not intended to be user-interactive. Normally index.jsp can be safely deleted. However, our sample service's welcome page contains instructions on how to call the servlet using `wget`, so in this case it should not be deleted.

The name of the war-file (web archive) is WebLichtSampleService.war, and WebLichtSampleService is the context, so it is the first part of the path after the host name. The context name can be modified in META-INF/context.xml.

A servlet within the WebLichtSampleService context called lemmaServlet03 is in a source package called servlets. The mappings between servlet names and their corresponding class names can be specified in WEB-INF/web.xml. Our sample service is available at:

http://localhost:8080/WebLichtSampleService/lemmaServlet03

You can test the service with the sample input file from the command line:

```
wget --post-file=sampleInput.xml http://localhost:8088/WebLichtSampleService/lemmaServlet03 -O
lemmaServlet03Out.xml
```

The output of lemmaServlet03 will be stored in the file lemmaServlet03Out.xml.

10.2.3.2 A Look at the Servlet Code

A servlet is just a Java class that extends HttpServlet, which contains `doGet` and `doPost` methods[11] that can be overridden to perform GET and POST requests, respectively. Each of these methods has a request and a response object as parameters.

The `doPost` method first sets the response's contentType to „text/xml" with „UTF-8" character encoding, which is TCF standard. Next, the document that has been posted is read from the request's input stream into a temporary file. An input stream from this temporary file, along with the response's output stream is then sent to the `LemmaSummarizer` object, which does the real work of the service. Finally, the temporary file is deleted and the response's output stream is closed. If any errors are encountered, an error is sent to the response with an appropriate message. Note that although it may seem unnecessary to store the input document in a file before processing, it is more reliable than reading directly from the request's input stream.

Our `doGet` method simply sends an error to the response because we are expecting data to be sent with the POST method.

10.2.3.3 A Look at the LemmaSummarizer

The `LemmaSummarizer` object does all of the real work of the service. It simultaneously parses the input stream (posted document) and writes all necessary data to the response's output stream. All elements, including their attributes, are read from the input document are copied to the response. All information needed to perform the service's task is saved during parsing. After all of the TextCorpus layers have been parsed and copied to the response stream, but before the TextCorpus end tag is written, the new layer is calculated an written to the response. Then the remaining end tags are written.

The only public method in `LemmaSummarizer` is `summarizeLemmas`. It creates a parser for the input stream, a writer for the output stream, and a map to store Token objects with the token id as key. When the TextCorpus element is encountered, it is processed by the processTextCorpus method.

The `processTextCorpus` method parses and writes the TextCorpus layer, processing the lemmas layer with the `processLemmas` method and the tokens layer with the `processTokens` method. These two methods fill in the tokenIdMap, so that by the time the TextCorpus end element is reached, the map is filled with Token objects. For our sample input, the map looks like this:

t1	lemma: Karin token: Karin
t2	lemma: essen token: aß
t3	lemma: Pizza token: Pizza
t4	lemma: . token: .
t5	lemma: Max token: Max

[11] This pertains to Java EE 5. There are some simplifications in the upcoming Java EE 6.

t6	lemma: haben token: hat
t7	lemma: Pizza token: Pizza
t8	lemma: essen token: gegessen
t9	lemma: . token: .
t10	lemma: Sie|sie|sie token: Sie
t11	lemma: essen token: essen
t12	lemma: Pizza token: Pizza
t13	lemma: . token: .

Before the end TextCorpus element is written to the response, the new layer is added in the `writeLemmaSummary` method. This method manipulates the map above and creates a new map with the lemmas as keys and a list of TokenCount objects as values. The result for our sample data is:

essen	aß	essen	gegessen
	1	1	1
Karin	Karin		
	1		
Max	Max		
	1		
Sie|sie|sie	Sie		
	1		
.	.		
	3		
haben	hat		
	1		
Pizza	Pizza		
	3		

It can be seen from this map that the input document contains the lemma “essen” a total of three times: once as the token “aß”, once as the token “essen” and once as the token “gegessen”. The lemma “Pizza” appears three times as token “Pizza”.

Once this map has been produced, it is simply iterated and written to the response as well-formed xml.

10.2.3.4 Why StAX?

If you are familiar with DOM parsing, you may ask why a StAX parser is being used in this sample service tutorial. DOM parsers hold an entire document in memory, so elements can be easily

accessed and added. It is perfectly suited to our needs, except for its one major drawback – it cannot efficiently process large documents, and may even cause an OutOfMemoryException. A StAX parser on the other hand, is very fast and efficient, and generally will not cause memory problems even with very large documents because the programmer decides which pieces of information are saved and how they are saved.

10.2.4 Schemas

The structure of TCF files is defined in the schemas dspin_0_3.xsd and dspin_textcorpus_0_3.xsd. The file dspin_0_3.xsd is just a skeleton for defining the metadata and includes the file dspin_textcorpus_0_3.xsd. Besides the textCorpus format, other linguistic data formats can be specified, including lexicon or dictionary formats. Currently, TCF concentrates on the main layers of linguistic annotations.

10.2.5 Adding a Service to WebLicht

Once you have developed a service and made it available, you will need to register it in the WebLicht repository, which will make it available in the WebLicht web application. You may also need to have the schemas changed.

- If you are adding a layer that already exists in the schema (a "tokens" layer, for example), please use the same xml output format that is used by existing services producing that layer.
- If you are adding a layer that does not yet exist in the schema, the D-SPIN development team will need to add the new layer to the schema.

For information about changing the schema and/or registering new services in WebLicht, please contact the D-SPIN development group at: weblicht@d-spin.org

10.3 Tutorial for Implementing WebLicht Web Services in Perl

10.3.1 Introduction

This tutorial describes the implementation of WebLicht web services in Perl on Linux systems using a DOM parser. Familiarity with the Perl programming language, XML documents, and WebLicht is assumed. This tutorial can be downloaded from http://weblicht.sfs.uni-tuebingen.de/weblichttutorial.shtml

10.3.2 Wrapper Example 1: txt2tcf.perl

We will start with a first example of a wrapper program, namely the **txt2tcf.perl** web service implemented at the University of Stuttgart. It converts a plain text document into a TCF document with UTF-8 encoding. Using the **wget** program, which is available on most Linux systems, the web service can be called as follows to convert the plain text file **test.txt** into the TCF document **test.xml**:

```
> wget --post-file=test.txt http://gelbaugenpinguin.ims.uni-stuttgart.de/cgi-
bin/dspin/txt2tcf3.perl --output-document=test.xml
```

The wrapper reads the input file, guesses the character encoding, converts to Unicode, generates an XML document, and prints it.

We will now look line by line at the actual code in order to understand how the program works. The complete code is listed in

Appendix A.

The first line of the wrapper program tells the operating system the location of the Perl interpreter, which is to be used to execute this script:

```
#!/usr/bin/perl
use Encode;
use Encode::Guess;
use XML::LibXML;
```

The **Encode** module is needed to convert strings between different character sets. **Encode::Guess** provides a character set guesser, and the **XML::LibXML** module implements a Perl interface to the efficient XML parser **libxml2**, which is implemented in C++. A documentation of the **XML::LibXML** module is available at http://search.cpan.org/dist/XML-LibXML.

libxml2 is a DOM parser, which reads an XML file into an internal data structure for further processing and manipulation. DOM parsers are more comfortable to work with than SAX parsers which process the XML file without building an internal data structure by calling user-defined processing function via hooks whenever a predefined XML element is parsed. The disadvantage of DOM parsers is their bigger demand for memory. **libxml2** is an efficient DOM parser, which is able to process large XML files.

The next two lines define variables for the name of the top element of the TCF document (D-SPIN) and the encoding (UTF-8) of the XML file that will be produced as output.

```
my $header = 'D-Spin';
my $encoding = 'UTF-8';
```

Now the namespaces used in the output document are defined.

```
my $def = 'http://www.dspin.de/data';
my $tc = 'http://www.dspin.de/data/textcorpus';
my $md = 'http://www.dspin.de/data/metadata';
```

The whole input file is read into the variable **$in**.

```
my $in;
{ local $/; $in = <>; }
```

The command `#local $/;#` temporarily undefines the input record separator in order to allow the following command `#$in = <>#` to read in the whole file at once.

Now the guesser is called to determine the encoding of the file.

```
my $enc = guess_encoding($in);
$enc = guess_encoding($in, 'latin1') unless ref($enc);
```

By default, it considers ASCII and the different Unicode variants. If none of these applies, the guesser is called again to check whether the input is compatible with a Latin1 encoding. If not, an error is reported by calling the function **report_error**:

```
report_error(1,"invalid input","unable to guess encoding") unless ref($enc);
```

The next command decodes the input string into Perl's internal string format:

```
$in = $enc->decode($in);
```

A new XML document with the root element **D-Spin** is created with the namespace stored in **$def**.

```
my $doc = XML::LibXML->createDocument('1.0', $encoding);
my $root = $doc->createElementNS( $def, "D-Spin" );
```

A version attribute, an empty **MetaData** child element, and a **TextCorpus** child element are added:

```
$root->addChild($doc->createAttribute('version', '0.3'));
$root->addNewChild( $md, 'MetaData' );
my $corpus = $root->addNewChild( $tc, 'TextCorpus' );
```

Finally, a child element **text** is added to the **TextCorpus** element and the input file is stored as its content.

```
$doc->setDocumentElement( $root );
my $text = $corpus->addNewChild($tc, 'text');
$text->addChild($doc->createTextNode($in));
```

Now we are ready to print the HTTP header and the resulting XML document:

```
print "Content-Type: text/xml; charset=$encoding\n\n";
print $doc->toString(1);
```

The argument of the libxml2 serialization function **toString** turns on the pretty print mode.

Whenever the wrapper encounters an error, it calls the function **report_error**, which prints an HTTP header and an XML document with an error report and terminates.

```
sub report_error {
    my $id = shift;
    my $code = shift;
    my $message = shift;
    print "Content-Type: text/xml; charset=utf-8
Status: 400 bad request

<?xml version=\"1.0\" encoding=\"utf-8\"?>
<$header xmlns=\"http://www.dspin.de/data\" version=\"0.3\">
 <error xmlns=\"http://www.dspin.de/data/error\">
  <type code=\"$id\">$code</code>
  <message>$message</message>
 </error>
</$header>
";
    die "Error: $message!";
}
```

10.3.3 Wrapper Example 2: tree-tagger.perl

The **tree-tagger.perl** wrapper implements a POS tagging web service for different languages based on the TreeTagger software. The input document is a TCF document with **tokens**. The wrapper returns the same document with the two additional elements **POStags** and **lemmas**. The TreeTagger has to read a large parameter file when it is started. This takes some time. Therefore the wrapper connects to a TreeTagger **daemon** that resides in memory. The wrapper and the tagger daemon communicate via sockets. The complete wrapper code can be found in

Appendix B.

The TreeTagger wrapper uses the following Perl modules:

```
use Encode;
use IO::Socket;
use XML::LibXML;
use XML::LibXML::XPathContext;
```

The **IO::Socket** module is required to create a connection to the TreeTagger daemon. **XML::LibXML::XPathContext** is needed to execute XPath expressions that extract the **token** elements from the TCF document.

The next commands define the name of the header element and the required name spaces.

```
my $header = 'D-Spin';
my $def = 'http://www.dspin.de/data';
my $tc = 'http://www.dspin.de/data/textcorpus';
```

Now we define mappings between languages and port numbers and between languages and tag sets. There is a separate tagger daemon for each language and each daemon is associated with a port.

```
my($in,%port,%tagset);
$port{'en'} = 7070;
$port{'de'} = 7071;
$port{'fr'} = 7072;
$port{'it'} = 7073;

$tagset{'en'} = "PennTB";
$tagset{'de'} = "STTS";
$tagset{'fr'} = "SteinFR";
$tagset{'it'} = "SteinIT";
```

The input XML file is read and parsed:

```
{ local $/; $in = <>; }

my $parser = XML::LibXML->new();
my $dom = $parser->parse_string( $in );
```

An error is reported if the parsing failed:

```
report_error(1, "invalid input", "unable to parse XML document")
    unless defined $dom;
```

The encoding of the input document is stored in a variable.

```
my $encoding = $dom->encoding();
```

An **XPathContext** is created for the document and the two namespaces are registered.

```
my $xc = XML::LibXML::XPathContext->new( $dom->documentElement() );
$xc->registerNs( 'def', $def );
$xc->registerNs( 'tc', $tc );
```

The following loop finds all TextCorpus elements (although currently only one TextCorpus element is allowed in TCF documents). The XPath expression `#/*/tc:TextCorpus#` matches any **TextCorpus** child elements of the top-most node (which is named D-Spin).

```
my $corpus;
foreach $corpus ($xc->findnodes( '/*/tc:TextCorpus' )) {
```

The language attribute of the **TextCorpus** is retrieved. If it is not one of the covered languages, nothing is done.

```
my $lang = $corpus->getAttribute( 'lang' );
    next unless exists $port{$lang};
```

The **tokens** element is searched inside the **TextCorpus** element. If there is none, the wrapper reports an error and terminates.

```
report_error(1,"invalid input","no input tokens")
        unless $xc->findnodes( 'tc:tokens' ,$corpus);
```

Now the list of tokens is extracted.

```
my @tokens = $xc->findnodes( 'tc:tokens/tc:token' ,$corpus);
```

A connection to the socket of the tagger daemon for the respective language is created. An error is reported if the connection could not be established.

```
my $sock = new IO::Socket::INET( PeerAddr => 'localhost',
                                  PeerPort => $port{$lang},
                                  Proto => 'tcp');
    report_error(2, "internal error", "unable to connect to socket")
        unless $sock;
```

The token strings are extracted and sent to the tagger in UTF-8 encoding with one token per line. Empty tokens are reported as an error.

```
my $token;
    foreach $token (@tokens) {
        my $tok = $token->textContent;
        report_error(1, "invalid input", "empty token in XML document")
            if $tok eq '';
        print $sock encode("utf-8", $tok),"\n";
    }
```

The input connection to the tagger daemon is closed.

```
shutdown($sock, 1);
```

Two new XML elements **POStags** and **lemmas** are added to the **TextCorpus** element. and the tagset attribute of the **POStags** element is properly defined.

```
my $POStags = $corpus->addNewChild($tc, 'POStags');
    my $lemmas = $corpus->addNewChild($tc, 'lemmas');
    $POStags->addChild($dom->createAttribute('tagset', $tagset{$lang}));
```

Now the output of the tagger daemon is read. There is one line of output for each input token. We iterate over the input tokens, read one line for each token, and decode it from UTF-8 format. An error is reported if nothing was read.

```
foreach $token (@tokens) {
        $_ = decode("utf-8", <$sock>);
        report_error(2, "internal error", "erroneous socket output")
            unless defined $_;
```

The newline character is stripped and the input is split at tab character positions. The elements are stored in the variables **$w**, **$t**, and **$l**.

```
chomp;
        my($w,$t,$l) = split(/\t/);
```

The token ID is retrieved and an error is reported if it cannot be found.

```
my $id = $token->getAttribute('ID');
        report_error(1, "invalid input", "missing token ID in XML document")
            unless defined $id;
```

A **tag** element is added to **POStags** for each **token**.

```
add_child( $POStags, 'tag', $id, $t);
```

Similarly, a **lemma** element is added to **lemmas** for each **token**.

```
add_child( $lemmas, 'lemma', $id, $l);
```

The socket connection is closed.

```
close $sock;
```

Finally the result document is printed.

```
print "Content-Type: text/xml; charset=$encoding\n\n";
print $dom->toString(1);
```

The following auxiliary function adds an element whose name is the second argument to the node transmitted as the first argument, and stores the token ID as the value of the attribute **tokenIDs**, and the text string as the text content of the element.

```
sub add_child {
    my $node = shift;
    my $name = shift;
    my $id = shift;
    my $text = shift;

    my $child = $node->addNewChild($tc, $name);
    $child->addChild($dom->createAttribute('tokenIDs', $id));
    $child->addChild($dom->createTextNode($text));
}
```

10.3.4 Testing

The above wrappers can be tested by calling them with an appropriate argument file. The input can also be “piped” into the program as shown below:

```
> echo "This is a test." | ./txt2tcf.perl
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
<D-Spin xmlns="http://www.dspin.de/data" version="0.3">
  <MetaData xmlns="http://www.dspin.de/data/metadata"/>
  <TextCorpus xmlns="http://www.dspin.de/data/textcorpus">
    <text>This is a test.</text>
  </TextCorpus>
</D-Spin>
```

The tagger web service can be tested with the command

```
> ./tree-tagger.perl test.xml
```

where test.xml contains the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<D-Spin xmlns="http://www.dspin.de/data" version="0.3">
  <MetaData xmlns="http://www.dspin.de/data/metadata"/>
  <TextCorpus xmlns="http://www.dspin.de/data/textcorpus" lang="en">
    <text>This is a test.</text>
    <tokens>
      <token ID="t1">This</token>
      <token ID="t2">is</token>
      <token ID="t3">a</token>
      <token ID="t4">test</token>
      <token ID="t5">.</token>
    </tokens>
    </sentences>
  </TextCorpus>
</D-Spin>
```

10.3.5 Installation

The installation depends on the type of web server. In case of an Apache server, the wrapper programs need to be copied to the directory **/www/cgi-bin**.

In order to be able to use the web service in WebLicht, you have to register it in the web service registry in Leipzig.

Appendix A – Wrapper Example 1: txt2tcf.perl

```
#!/usr/bin/perl -w

use Encode;
use Encode::Guess;
use XML::LibXML;

my $header = 'D-Spin';
my $encoding = 'UTF-8';

my $def = 'http://www.dspin.de/data';
my $tc = 'http://www.dspin.de/data/textcorpus';
my $md = 'http://www.dspin.de/data/metadata';

my $in;
{ local $/; $in = <>; }

my $enc = guess_encoding($in);
$enc = guess_encoding($in, 'latin1') unless ref($enc);
report_error(1,"invalid input","unable to guess encoding") unless ref($enc);
$in = $enc->decode($in);

my $doc = XML::LibXML->createDocument('1.0', $encoding);
my $root = $doc->createElementNS( $def, "D-Spin" );
$root->addChild($doc->createAttribute('version', '0.3'));
$root->addNewChild( $md, 'MetaData' );
my $corpus = $root->addNewChild( $tc, 'TextCorpus' );
$doc->setDocumentElement( $root );
my $text = $corpus->addNewChild($tc, 'text');
$text->addChild($doc->createTextNode($in));

# generate the output
print "Content-Type: text/xml; charset=$encoding\n\n";
print $doc->toString(1);


sub report_error {
    my $id = shift;
    my $code = shift;
    my $message = shift;
    print "Content-Type: text/xml; charset=utf-8
Status: 400 bad request

<?xml version=\"1.0\" encoding=\"utf-8\"?>
<$header xmlns=\"http://www.dspin.de/data\" version=\"0.3\">
 <error xmlns=\"http://www.dspin.de/data/error\">
  <type code=\"$id\">$code</code>
  <message>$message</message>
 </error>
</$header>
";
    die "Error: $message!";
}
```

Appendix B – Wrapper Example 2: tree-tagger.perl

```
#!/usr/bin/perl -w

use Encode;
use IO::Socket;
use XML::LibXML;
use XML::LibXML::XPathContext;

my $header = 'D-Spin';

my $def = 'http://www.dspin.de/data';
my $tc = 'http://www.dspin.de/data/textcorpus';

# define the port numbers of the tagger daemons for the different languages
my($in,%port,%tagset);
$port{'en'} = 7070;
$port{'de'} = 7071;
$port{'fr'} = 7072;
$port{'it'} = 7073;

$tagset{'en'} = "PennTB";
$tagset{'de'} = "STTS";
$tagset{'fr'} = "SteinFR";
$tagset{'it'} = "SteinIT";

# read the whole input at once and then restore line-by-line reading
{ local $/; $in = <>; }

# parse the XML input
my $parser = XML::LibXML->new();
my $dom = $parser->parse_string( $in );
report_error(1, "invalid input", "unable to parse XML document")
    unless defined $dom;

my $encoding = $dom->encoding();

my $xc = XML::LibXML::XPathContext->new( $dom->documentElement() );
$xc->registerNs( 'def', $def );
$xc->registerNs( 'tc', $tc );

# process all input corpora
my $corpus;
foreach $corpus ($xc->findnodes( '/*/tc:TextCorpus' )) {
    my $lang = $corpus->getAttribute( 'lang' );
    next unless exists $port{$lang};

    report_error(1,"invalid input","no input tokens")
        unless $xc->findnodes( 'tc:tokens' ,$corpus);

    # extract the tokens that are to be annotated
    my @tokens = $xc->findnodes( 'tc:tokens/tc:token' ,$corpus);

    # create a socket connection to the tagger daemon
    my $sock = new IO::Socket::INET( PeerAddr => 'localhost',
                                     PeerPort => $port{$lang},
                                     Proto => 'tcp');
    report_error(2, "internal error", "unable to connect to socket")
        unless $sock;

    # send the tokens to the tagger daemon
    my $token;
    foreach $token (@tokens) {
        my $tok = $token->textContent;
        report_error(1, "invalid input", "empty token in XML document")
            if $tok eq '';
        print $sock encode("utf-8", $tok),"\n";
    }
    shutdown($sock, 1);

    # add new XML elements for the tags and the lemmas
    my $POStags = $corpus->addNewChild($tc, 'POStags');
    my $lemmas = $corpus->addNewChild($tc, 'lemmas');
    $POStags->addChild($dom->createAttribute('tagset', $tagset{$lang}));

    # read the POS tags and lemmas from the tagger daemon
    foreach $token (@tokens) {
        $_ = decode("utf-8", <$sock>);
        report_error(2, "internal error", "erroneous socket output")
```

```
            unless defined $_;
        chomp;
        my($w,$t,$l) = split(/\t/);
        my $id = $token->getAttribute('ID');
        report_error(1, "invalid input", "missing token ID in XML document")
            unless defined $id;

        # Add a tag element to the document
        add_child( $POStags, 'tag', $id, $t);

        # Add a lemma element to the document
        add_child( $lemmas, 'lemma', $id, $l);
    }
    close $sock;
}

# generate the output
print "Content-Type: text/xml; charset=$encoding\n\n";
print $dom->toString(1);

sub add_child {
    my $node = shift;
    my $name = shift;
    my $id = shift;
    my $text = shift;

    my $child = $node->addNewChild($tc, $name);
    $child->addChild($dom->createAttribute('tokenIDs', $id));
    $child->addChild($dom->createTextNode($text));
}

sub report_error {
    my $id = shift;
    my $code = shift;
    my $message = shift;
    print "Content-Type: text/xml; charset=utf-8
Status: 400 bad request

<?xml version=\"1.0\" encoding=\"utf-8\"?>
<$header xmlns=\"http://www.dspin.de/data\" version=\"0.3\">
 <error xmlns=\"http://www.dspin.de/data/error\">
  <type code=\"$id\">$code</code>
  <message>$message</message>
 </error>
</$header>
";
    die "Error: $message!";
}
```

Appendix C – List of Resources

Resources registered by D-SPIN project partners at the CLARIN Virtual Language Observatory

Name	Language	Type	Description	Partner
GermaNet	German	Lexicon / Knowledge Source	61 600 synsets, 84 500 lexical units, XML	University of Tübingen
Tübingen Partially Parsed Corpus of Written German – TüPP-D/Z	German	Treebank	Automatically annotated corpus, treebank, written language, newspaper texts; 200 million tokens; Part of speech, chunks, topological fields	University of Tübingen
Tübingen Treebank of Spoken English – TüBa-E/S	English	Treebank	Semi-automatic annotation, oral text/ dialogues / corpus, 30 000 sentences, 310 000 tokens, Morphology, part of speech, syntax (functional and dependency information)	University of Tübingen
Tübingen Treebank of Spoken German – TüBa-D/S	German	Treebank	Semi-automatic annotation, oral text/ dialogues / corpus, 36 000 sentences, 380 000 Tokens, Morphology, part of speech, syntax (functional and dependency information), topological fields, named entities, coreference relations	University of Tübingen
Tübingen Treebank of Spoken Japanese – TüBa-J/S	Japanese	Treebank	Semi-automatic annotation, oral text/ dialogues / corpus, 18 000 sentences, 160 000 Tokens, Morphology, part of speech, syntax (functional and dependency information)	University of Tübingen
Tübingen Treebank of Written German – TüBa-D/Z	German	Treebank	Semi-automatic annotation, written corpus, newspaper corpus, 45,200 sentences with 794,000 Tokens, Morphology, part of speech, syntax (functional and dependency information), topological fields, named entities, coreferential annotation	University of Tübingen
Bilingual Language Acquisition Julka Corpus	German Polish	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
Code-switching conversation corpus	Dutch	Spoken Corpus	The code-switching corpus consists of 5x30-minute conversations between four speakers (i.e. a total of 20 speakers). The speakers are bilingual speakers of Papiamento (a creole language spoken in the Dutch Antilles) and Dutch. In the course of their free conversations, they engage in code switching, that is, they use both languages within the same utterance in systematic ways. The corpus is fully transcribed and glossed, coded for language and word class, in ELAN.	Max Planck Institute for Psycholinguistics
Dutch Bilingualism Data Base (DBD)	Arabic Dutch	Spoken Corpus	Audio recordings, transcripts,	Max Planck Institute for Psycholinguistics

	Turkish			
ECHO corpus	Dutch British Swedish German	Multimodal Corpus	This is a corpus of four European sign languages. It contains richly annotated video files of Sign Language of the Netherlands (Nederlandse Gebarentaal), British Sign Language, and Swedish Sign Language; data include narratives, dialogues, small lexicons, and poetry. In addition, parts of a corpus of German Sign Language (Deutsche Gebärdensprache) is included that was already published on paper before.	Max Planck Institute for Psycholinguistics
L1 & L2 Acquisition Christine Dimroth German Project	German	Spoken Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 & L2 Acquisition Marzena Watorek French Project	French Polish	Spoken Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Anke Jolink	Dutch	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Barbara Schmiedtova	German	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Bhuvana Narasimhan	Hindi Tamil	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Carmel O'Shannessy	Warlpiri	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Caroline Corpus	German	Spoken Corpus Written Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Chen Jidong	Chinese	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Christine Dimroth & Bhuvana Narasimhan	German	Spoken Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Gaby Cablitz	French	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Joost van de Weijer	Dutch	Spoken Corpus Written Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Max Miller	German	Spoken Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Penelope Brown Rossel	English	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Penelope Brown Tzeltal	Tenejapa Tzeltal	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L1 Acquisition Stoll	Russian	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L2 Acquisition Barbara Schmiedtova	Czech English German Vietnamese	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L2 Acquisition Christine Dimroth	Croatian German	Spoken Corpus Written	Language Acquisition corpus	Max Planck Institute for Psycholinguistics

	Russian Turkish	Corpus		
L2 Acquisition Finiteness and Scope	Dutch English French German	Spoken Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L2 Acquisition Heide Wegener	German	Spoken Corpus Written Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L2 Acquisition P-Moll Norbert Dittmar	German Italian Polish	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
L2 Acquisition Ursula Stephany & Christine Dimroth	German	Spoken Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
LAC Akuriyo Corpus	Akuriyo	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Analysis and description of the Tiriyó language	Tiriyó	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Chontal Corpus	Lowland Chontal	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Cusco Quechua Corpus	Quechua	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC English Corpus	English	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Ewe Corpus	Ewe	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Goemai Corpus	Goemai	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Guugu Yimithirr	Guugu Yimithirr	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Hindi Corpus	Hindi	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Inuktitut Corpus	Inuktitut	Written Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Italian Corpus	Italian	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Jahai Corpus	Jahai	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Jalonke Corpus	Jalonke	Spoken Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Jaminjung/Ngaliwurru Corpus	Jaminjung Ngaliwurru	Spoken Corpus Written Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Karihona Corpus	Karihona	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Katang Corpus	Katang	Spoken Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Khmer Corpus	Khmer	Spoken Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Kilivila Corpus	Kilivila	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Kri Corpus	Kri	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics

LAC Lao Corpus	Lao	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Lavukaleve Corpus	Lavukaleve	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Mawe Corpus	Mawe	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Miranya Corpus	Miranya	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Nqeq Corpus	Nqeq	Spoken Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Savosavo Corpus	Savosavo	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Tidore Corpus	Tidore	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Tzeltal Corpus	Tzeltal	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Umpithamu Corpus	Umpithamu	Spoken Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
	Umbuygamu	Written Corpus		
LAC Yucatec Corpus	Yucatec	Multimodal Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
LAC Zapotec and Chatino Survey	Zapotec	Spoken Corpus, Written Corpus	Language and Cognition corpus	Max Planck Institute for Psycholinguistics
	Chatino			
Marquesan corpus	Macedonian	Multimodal Corpus	Documentation of the Marquesan language and culture project (DoBeS project)	Max Planck Institute for Psycholinguistics
MPI Language and Cognition Corpus	many different languages including endangered languages	Multimodal Corpus	Growing corpus about many different languages including endangered languages; almost perfect structure for IEI; completely metadata described; containing video, sound and annotations	Max Planck Institute for Psycholinguistics
Multilingualism Marianne Gullberg & Peter Indefrey	French	Multimodal Corpus	Language Acquisition corpus	Max Planck Institute for Psycholinguistics
	Dutch			
	German			
	English			
Semang corpus	Batek Deq	Multimodal Corpus	Documentation of the Semang project (DoBeS project)	Max Planck Institute for Psycholinguistics
	Jahai			
	Lanoh			
	Maniq			
	Menriq			
	Semnam			
Virtual Language Observatory	English	other	KMZ overlay for GoogleEarth containing placemarks/links to CLARIN members, Tools, Resources and other archives/sites.	Max Planck Institute for Psycholinguistics
Yurakaré corpus	Yurakaré	Multimodal Corpus	Documentation of the Yurakaré project (DoBeS project)	Max Planck Institute for Psycholinguistics
≠Akhoe Hai//om	≠Akhoe Hai//om	Multimodal Corpus	Documentation of the ≠Akhoe Hai//om project (DoBeS project)	Max Planck Institute for Psycholinguistics

Archiv für Gesprochenes Deutsch (AGD)	German	Spoken Corpus	Recordings of dialects, conversations, institutional interaction. AGD ("Archiv für gesprochenes Deutsch", in former times called "Deutsches Spracharchiv") archives german spoken corpora of research projects and allocates them for further scientific research. Available digitized soundfiles and transcripts are researchable in DGD ("Datenbank gesprochenes Deutsch").	IDS
Bibliografie zur deutschen Grammatik (BDG)	German	other	Online Bibliography, bibliographic database	IDS
Cyril Belica : Kookkurrenzdatenbank CCDB	German	other	A cooccurrence database developed by the Institut für Deutsche Sprache, for research in the field of collocation analysis in modern German. The database holds over 200,000 analyzed words that can be browsed or searched and shown in context.	IDS
Datenbank gesprochenes Deutsch (DGD)	German	Spoken Corpus	Recordings of dialects, conversations, institutional interaction. DGD is the researchable online-database of the AGD: documentary information of nearly all corpora, approx. 8600 digitized recordings of 2100 hours; 3100 transcripts thereof 3000 sound-aligned; transcripts and sound files are downloadable after registration	IDS
Deutsches Referenzkorpus (DeReKo)	German	Written Corpus	Written general monolingual synchronic (1959-) reference corpus archive; 3.9 billion tokens; structural information down to sentence level, rich bibliographic metadata, partial layout information, fully morpho-syntactically annotated	IDS
Diskurswörterbuch zur Umbruchsgeschichte der Nachkriegszeit (im elexiko-Portal)	German	Lexicon / Knowledge Source	Ca. 85 entries (plus about 200 sublemmata) XML	IDS
elexiko-Wörterbuch (im elexiko-Portal)	German	Lexicon / Knowledge Source	XML	IDS
Gesprächanalytisches Informationssystem (GAIS)	German	other	Web-based information system on scientific community (news, events, persons, job market, mailing list, database on research projects and corpora, bibliography, glossary and links) and recording equipment/software; disciplinary scope: research on conversation and discourse analysis and spoken language	IDS

Grammatisches Informationssystem (grammis)	German	Lexicon / Knowledge Source	Web Information System – contains e.g. a linked terminological knowledge-base, XML format	IDS
Historisches Korpus	German	Written Corpus	Written, general, diachronic, monolingual; 70 m. (currently being compiled)	IDS
Online Bibliography of Electronic Lexicography (OBELEX)	German	Other	OBELEX includes bibliographic information about articles, monographs, anthologies and reviews published since 2000 that relate to electronic lexicography, as well as some relevant older works; the particular focus is on works about online lexicography. In the OBELEX bibliography, all relevant research contributions are systematically compiled and are searchable by different criteria such as keyword, language, author, title, and publication year.	IDS
Propädeutische Grammatik (ProGr@mm)	German	Grammar	Web Information &Learning System, contains reusable e-learning resources, XML format	IDS
Wörterbuch der Neologismen der 90er Jahre (im elexiko-Portal)	German	Lexicon / Knowledge Source	Ca. 700 entries, XML	IDS
Berliner Wendecorpus	German	Spoken Corpus Web Service	Transcribed narrative interviews with people from East and West Berlin about the events of November 9. 282,000 tokens. TEI XML, lemma and POS. Normalized version also available.	BBAW
DDR-Korpus	German	Written Corpus	9 million words in 1150 texts from GDR written between 1949 and 1990. Part of the DWDS project	BBAW
DWDS Kernkorpus	German	Spoken Corpus Written Corpus	German reference corpus. Ca. 100 million words, 20th Century. Searchable online. Part of 'Digitales Wörterbuch der deutschen Sprache des 20. Jahrhunderts' project	BBAW
Etymologisches Wörterbuch des Deutschen	German	Lexicon / Knowledge Source	Etymological dictionary of the German language	BBAW
Juilland-D-Korpus	German	Written Corpus	Written German from 1920-39. 500,000 tokens, 392 texts. POS and lemma, TEI XML. Part of Das digitale Wörterbuch der deutschen Sprache der 20. Jahrhunderts	BBAW
Korpus Berliner Zeitung	German	Written Corpus	Articles from the 'Berliner Zeitung' online edition from 3.1.1994 to 31.12.2005. About 252 million tokens in 869,000 articles. Part of the DWDS project.	BBAW
Korpus Gesprochene Sprache	German	Spoken Corpus	Transcribed speech from the 20th Century, about 2,5 million words. 7 categories, 756 speakers. Part of Digitales Wörterbuch der deutschen Sprache des 20. Jahrhunderts project	BBAW

Wörterbuch der deutschen Gegenwartssprache (WDG)	German	Lexicon / Knowledge Source	6 volume dictionary of Standard German, retro-digitization of the printed version which appeared 1964-1977, ca. 130 000 headwords	BBAW
ZEIT-Corpus	German	Written Corpus	Corpus of the weekly Die Zeit from 1946 - present day (complete runs from 1996). Over 100 million words in 200,000 articles. Updated daily. Part of DWDS project.	BBAW
Chintang / Puma corpus	Chintang	Multimodal Corpus	Documentation of the Chintang / Puma project (DoBeS project)	University of Leipzig
	Puma			
Wortschatz	Afrikaans	Written Corpus	Collected from newspaper texts, web crawling, etc.: words (+frequency), cooccurrences (+graph), left/right neighbours, example sentences	University of Leipzig
	Albanian	Web Service		
	Bulgarian			
	Catalan			
	Chinese			
	Croatian			
	Czech			
	Danish			
	Dutch			
	English			
	Esperanto			
	Estonian			
	Finnish			
	French			
	German			
	Hungarian			
	Icelandic			
	Indonesian			
	Italian			
	Japanese			
	Korean			
	Latin			
	Latvian			
	Lithuanian			
	Malay			
	Norwegian			
	Occitan			
	Romanian			
	Russian			
	Slovak			
	Slovenian			
	Spanish			
	Sundanese			
	Swedish			
	Tagalog			

	Turkish			
	Vietnamese			
	Welsh			

References

Boehlke, V. (2010). A Generic Chaining Algorithm for NLP Webservices. *Web Services and Processing Pipelines in HLT Workshop, LREC 2010.*

Brants, T. (2000). TnT - A Statistical Part-of-Speech Tagger. In Proceedings of the . *Sixth Applied Natural Language Processing Conference ANLP-2000.* Seattle, WA.

Cahill, A. (2008). Treebank-Based Probabilistic Phrase Structure Parsing. *Language and Linguistics Compass , 2* (1), 18-40.

Church, K. W., & Hanks, P. (1990). Word association norms, mutual information, and lexicography. *Computational Linguistics , 16* (1), 22-29.

Fellbaum, C. (1998). *WordNet: An electronic lexical database.* Cambridge, MA: MIT Press.

Finkel, J. R., Grenager, T., & Manning, C. (2005). Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. *43nd Annual Meeting of the Association for Computational Linguistics (ACL 2005)*, (S. 363-370). Toulouse, France.

Grishman, R., & Sundheim, B. (1995). Design of the MUC-6 Evaluation. *Message Understanding Conference-6 (MUC-6).*

Hamp, B., & Feldweg, H. (1997). GermaNet - a Lexical-Semantic Net for German. *ACL workshop Automatic Information Extraction and Building of Lexical Semantic Resources for NLP Applications.* Madrid.

Heid, U., Schmid, H., Eckart, K., & Hinrichs, E. (2010). A corpus representation format for linguistic web services: the D-SPIN Text Corpus Format and its relationship with ISO standards. *Proceedings of LREC 2010.*

Henrich, V., Hinrichs, E., Hinrichs, M., & Zastrow, T. (2010). Service-Oriented Architectures: From Desktop Tools to Web Services and Web Applications. In D. T. Forăscu (Hrsg.), *Multilinguality and Interoperability in Language Processing with Emphasis on Romanian.* Bucharest: Romanian Academy Publishing House.

Petrov, S., Barrett, L., Thibaux, R., & Klein, D. (2006). Learning Accurate, Compact, and Interpretable Tree Annotation. *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics* (S. 433-440). Sydney, Australia: Association for Computational Linguistics.

Schiller, A., Teufel, S., & Thielen, C. (1995). *Guidelines für das Tagging deutscher Textcorpora mit STTS.* Tübingen: Universität Tübingen.

Schmid, H. (2000). *LoPar: Design and Implementation* . Stuttgart: Universität Stuttgart.

Schmid, H. (1993). Probabilistic Part-of-Speech Tagging Using Decision Trees. *International Conference on New Methods in Language Processing.*

Sokirko, A. (2003). DDC – A search engine for linguistically annotated corpora. *Proceedings of Dialogue 2003.* Protvino, Russia.

Ratnaparkhi, A. (1996). A Maximum Entropy Model for Part-Of-Speech Tagging. *Empirical Methods in Natural Language Processing Conference (EMNLP).* University of Pennsylvania.