The Technical Details of the D-Spin Architecture*

Helmut Schmid
IMS, University of Stuttgart

1 Background

D-Spin as a German subsidiary project of the European CLARIN project aims to develop a prototypical web service infrastructure for linguistic resources and tools. The main goals are to integrate a large number and diversity of linguistic resources (such as corpora, lexicons, thesauri etc.) as well as language processing tools (POS taggers, parsers, lemmatizers etc.) and to make these resources easily accessible to non-expert users.

D-Spin/CLARIN focusses on integrating existing resources rather than building new resources. These existing resources use their own idiosyncratic data representations and access methods and it is often difficult to combine them in order to build more complex applications. A translator, for instance, who wants to get an overview of the technical vocabulary used in some document that he or she wants to translate, could proceed as follows:

1. Obtain and install a tokenizer, a POS tagger, a lemmatizer, a term extraction tool and a statistics module.
2. Convert the document to plain text using a text processor.
3. Segment the text into sentences and words with the tokenizer.
4. Annotate the tokenized text with POS tags.
5. Lemmatize the tagged text.
6. Extract terms from the text (nouns, adjectives, and sequences of nouns and adjectives)
7. Compute terms which occur significantly more often in the document than in some large reference corpus (e.g. a newspaper corpus) which was preprocessed in the same way.

*Many thanks to Volker Boehlke, Thomas Zastrow, and Andreas Madsack for their constributions to this paper.

All these steps together would probably take at least a couple of days and would require programming skills in order to convert the data between the formats required by the different tools.

The D-Spin/CLARIN infrastructure will greatly simplify this task: CLARIN helps the user to locate available resources and services by means of catalogues. It is not necessary to install software or other resources because they are available as a web services. There is also no need to register or to obtain licenses for particular services since the CLARIN infrastructure knows the identity of its users and is able to decide whether access to a given resource should be granted or not (single sign-on). Finally, the user can easily combine different tools because the CLARIN resources use a standard interface and standard data representations.

Two important goals of the CLARIN project are therefore (i) to integrate a large number of existing resources and (ii) to make them interoperable by means of common data representation standards.

The integration of a wide range of existing resources demands active participation of the developers and/or expert users of these resources because the CLARIN project alone has not the required capacity. In order to allow for a large number of external developers to contribute to CLARIN, the communication overhead between developers and CLARIN administrators has to be kept at a minimum. Therefore the interfaces must be simple and easy to understand. The interfaces also have to be very flexible. Otherwise it will not be not possible to integrate new types of resources without costly modifications of the underlying CLARIN infrastructure.

The following sections propose an architecture for the prototypical webservice infrastructure to be implemented in D-Spin which is intended to meet these goals.

2 Web Service Infrastructure

The proposed architecture has four layers as shown in figure 1. The bottom layer consists of the existing software tools and resources which are to be integrated into the CLARIN framework. The lower intermediate layer is formed by (i) wrapper software which implements interfaces between existing tools and resources on the one hand, and the D-Spin/CLARIN framework on the other hand, (ii) converter tools which transform data from one representation into another and (iii) complex web services which recursively call simpler web services. The upper intermediate layer implements the "Webservice Interface". It authentificates the user, validates the format of the argument data passed with the webservice call (if there is any such data), calls the requested service (wrapper, converter, or complex service), and returns the results.

The top-most layer consists of application software which calls CLARIN webservices to perform various tasks. A special application is the "Chain Builder". The Chain Builder is an interactive software which helps the user to combine several CLARIN webservices in a processing pipeline to solve a task. The Chain Builder and the other

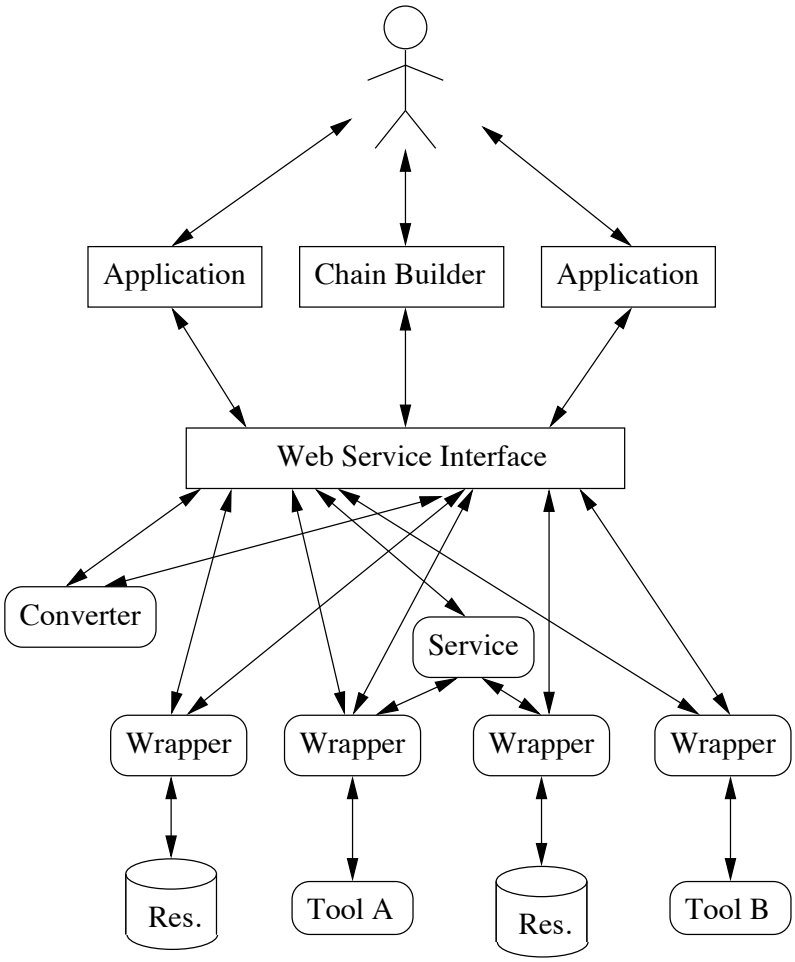

Figure 1: Overview of the D-Spin infrastructure

application programs are running on the user's computer either within a web browser or as a standalone program.

Application software makes it easy to use CLARIN webservices. Consider the CLARIN webservice for terminology extraction described in Fritzinger et al. 2009 (to appear). This webservice takes an XML-encoded corpus as input and returns another XML file with the extracted terms. In order to use this webservice, the corpus has to be converted to the correct XML format, the webservice has to be called and the results have to be presented in a readable form. All these task can be taken over by an application software. The software lets the user select a corpus with a file system browser, converts the file into a CLARIN document, sends the document to the terminology extraction webservice, retrieves the result and presents it to the user. It also allows the user to choose between various processing and display options. Without the help of such an application software, non-expert users will find it hard to use CLARIN webservices.

3 XML Data Exchange Format

All services in the intermediate layer receive an XML document as input and return another XML document which usually consists of the input document enriched with some additional information.

Each such CLARIN document contains a MetaData element[1] and a sequence of data elements. The following XML document is an example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<CLARIN>
 <MetaData>
  <source>IMS, Uni Stuttgart</source>
 </MetaData>
 <TextCorpus lang="de" text="yes">
  <text>Peter aß eine Pizza. Sie schmeckte ihm.</text>
 </TextCorpus>
</CLARIN>
```

This document could be the input to a tokenizer. The data element is a TextCorpus element which is to be tokenized. The output of the tokenizer might look as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<CLARIN>
 <MetaData>
  <source>IMS, Uni Stuttgart</source>
 </MetaData>
 <TextCorpus lang="de" text="yes" tokens="yes">
  <text>Peter aß eine Pizza. Sie schmeckte ihm.</text>
  <tokens>
   <token>Peter</token>
   <token>aß</token>
   <token>eine</token>
   <token>Pizza</token>
   <token>.</token>
   <token>Sie</token>
   <token>schmeckte</token>
   <token>ihm</token>
   <token>.</token>
  </tokens>
 </TextCorpus>
</CLARIN>
```

[1]The MetaData element is used by the top-level programs to store any information that they need to pass with the documents. This information is ignored by the service programs of the intermediate layer.

The output contains an additional complex element *tokens* and the additional attribute *tokens* at the TextCorpus element.

The attributes of data elements (such as the *TextCorpus* element in this example) characterise the contents of the data elements. The Process Chain Builder is able to decide solely on the basis of the data attributes whether some CLARIN service is applicable to a document or not. A German tokenizer e.g. might require documents containing one TextCorpus element with the attributes *lang="de"*, and *text="yes"*.

The input document requirements and output document properties of all CLARIN services are stored in a table. The I/O specification of an English part-of-speech tagger using the Penn Treebank tagset (PennTB) might look as follows:

```
<IO-Spec>
 <service>TreeTagger English</service>
 <input>
  <TextCorpus lang="en" tokens="yes"/>
 </input>
 <output>
  <TextCorpus lang="en" tokens="yes" POStagging="PennTB"/>
 </output>
</IO-Spec>
```

An important property of this architecture is that the top-level programs of the CLARIN architecture can deal with all the different services available through CLARIN at a very abstract level. All information that they need about the services can be stored in a table. The table contains the list of available web services and for each web service (i) its I/O specification, and (ii) the full path (or URL) of the program that needs to be executed.

3.1 XML Schemata

The formal structure of the CLARIN documents is defined in an XML schema which comprises a number of components, one for the MetaData element and one for each possible data type, such as the TextCorpus element in the above example. The XML schemata are used by the top-level programs to validate XML documents and they also serve as a documentation.

D-Spin will define a small number of XML schemata for frequently occurring data types such as corpora and lexicons, but it is clear that new web services will often require new data types. Therefore it is essential that new types can easily be added.

Appendix B shows a rudimentary XML schema for CLARIN documents in the Schema language RelaxNG. It might be useful to supplement such a RelaxNG schema with another schema in the Schematron language which is better suited to check certain

constraints. The Schematron schema shown in Appendix C, for instance, includes the following constraints:

If a *TextCorpus* element has the attribute *token_boundaries*[2], then

- an attribute *text* is required, too
- a *tokens* element must be present
- each *token* element must have a *start* and an *end* attribute
- the value of the *start* attribute must be smaller or equal to the value of the *end* attribute
- the value of the *end* attribute must not be larger than the length of the content of the *text* element
- the value of the *start* attribute of a *token* must be larger than the value of the end attribute of the preceding *token*.

4 Adding New Webservices

CLARIN's success will partly depend on how easy it is to add new webservices. If it is too difficult, CLARIN will not be able to attract enough external developers to integrate a wide range of resources.

Adding a new webservice to the CLARIN infrastructure proposed here involves the following steps:

- Definition of the input and output document formats

  In many cases, it will suffice to reuse existing document formats. In other cases, it might be necessary to extend an existing format or to define a new format.

- Implementation of the webservice software

  The software can either be implemented as a locally invoked program or as a CLARIN-internal webservice accessible via some URL.

- Submission of the new webservice to CLARIN

  The developer sends the name of the webservice, its I/O specification, its software with installation instructions (or a URL), and (optionally) new document type definitions in the form of XML schemata to the CLARIN administration which checks the submitted data and adds the new service.

[2]The attribute *token_boundaries* of a *TextCorpus* element indicates that the tokens are represented by their start and end positions in the content of the *text* element of the document.

5 Interoperability

The interoperability of the CLARIN webservices is guaranteed by their I/O specifications. If the output specification of some webservice A matches the input specification of some webservice B, then these two webservice can be chained together. If there is no perfect match, it might be possible to find a webservice which transforms the output of webservice A into a format which is suitable for webservice B.

In the following example, the output specification of the tokenizer does not match the input specification of the POS tagger.

```
<IO-Specs>
 <IO-Spec>
  <service>German tokenizer</service>
  <input>
   <TextCorpus lang="de"/>
  </input>
  <output>
   <TextCorpus lang="de" token_boundaries="yes"/>
  </output>
 </IO-Spec>

 <IO-Spec>
  <service>Token Converter</service>
  <input>
   <TextCorpus lang="de" token_boundaries="yes"/>
  </input>
  <output>
   <TextCorpus lang="de" token_boundaries tokens="yes"/>
  </output>
 </IO-Spec>

 <IO-Spec>
  <service>German tagger</service>
  <input>
   <TextCorpus lang="de" tokens="yes"/>
  </input>
  <output>
   <TextCorpus lang="de" tokens="yes" POStagging="STTS"/>
  </output>
 </IO-Spec>
</IO-Specs>
```

However, the CLARIN system might be able to infer that it is possible to transform the

tokenizer output to the tagger input format by applying the "Token Converter". The I/O specifications of the webservices are sufficient to draw these inferences.

The CLARIN system must be able to verify whether some document has the properties claimed by its attributes. Such a validation is necessary for documents supplied by the users and perhaps also for documents produced by newly added webservices which still have to prove their reliability. The validation is based on XML schemata. Validation against a RelaxNG schema will reject all documents with missing or unexpected elements and attributes or with values of the wrong type. More subtle inconsistencies could be detected with Schematron rules as explained above.

6 Advanced Issues

One important goal is to keep the D-Spin architecture as flexible as possible so that new web services can be added without changing the intrastructural software. An important question is therefore whether there are webservices which cannot be realised within the proposed framework. Here are some problems that the current architecture already can deal with.

6.1 Multiple inputs and outputs

The proposed architecture deals with multiple inputs or outputs by means of CLARIN documents containing multiple data elements. A context-free parser, for example, might take a document with a grammar and a tokenized corpus as input. The document might look as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<CLARIN>
 <MetaData>...</MetaData>
 <Grammar ...>... </Grammar>
 <TextCorpus ...>...</TextCorpus>
<CLARIN>
```

The top-level tools must be able to build such an XML document from two other documents. Similarly, they must be able to extract data elements from an XML document. However, they should never extract anything from within a data element, in order to avoid dependencies of the top-layer software on the document formats which might be changed in the future.

6.2 Program parameters

Some webservices might require additional parameters. An n-best parser, for instance, which returns the n best analyses of each sentence should be configurable wrt. parameter n. Such parameters could be supplied via an additional data element.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<CLARIN>
 <MetaData>...</MetaData>
 <NBestParserParameters>
  <N>50</N>
 </NBestParserParameters>
 <TextCorpus ...>...</TextCorpus>
<CLARIN>
```

A A Sample XML Document

This is an XML document containing a text corpus with tokenization, POS tagging, lemmatisation, constituent parses, dependency parses, and coreferences.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<CLARIN>

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

 <TextCorpus lang="de" text="yes" tokens="yes" token_boundaries="yes"
     sentences="yes" POStagging="STTS" lemmatisation="yes"
     coreferences="yes" parsing="TigerTB" depparsing="yes"
     source="ftp://www.ims.uni-stuttgart.de/pub/D-Spin/example.xml">
  <text>Peter aß eine Pizza. Sie schmeckte ihm.</text>
  <tokens>
   <token ID="t1" start="1" end="5">Peter</token>
   <token ID="t2" start="7" end="8">aß</token>
   <token ID="t3" start="10" end="13">eine</token>
   <token ID="t4" start="15" end="19">Pizza</token>
   <token ID="t5" start="20" end="20">.</token>
   <token ID="t6" start="22" end="24">Sie</token>
   <token ID="t7" start="26" end="34">schmeckte</token>
   <token ID="t8" start="36" end="38">ihm</token>
   <token ID="t9" start="39" end="39">.</token>
  </tokens>
  <sentences>
   <sentence ID="s1" start="1" end="20"/>
   <sentence ID="s2" start="22" end="40"/>
  </sentences>
  <POStags>
```

```
 <tag tokID="t1">NE</tag>
 <tag tokID="t2">VVFIN</tag>
 <tag tokID="t3">ART</tag>
 <tag tokID="t4">NE</tag>
 <tag tokID="t5">$.</tag>
 <tag tokID="t6">PPER</tag>
 <tag tokID="t7">VVFIN</tag>
 <tag tokID="t8">PPER</tag>
 <tag tokID="t9">$.</tag>
</POStags>
<lemmas>
 <lemma tokID="t1">Peter</lemma>
 <lemma tokID="t2">essen</lemma>
 <lemma tokID="t3">ein</lemma>
 <lemma tokID="t4">Pizza</lemma>
 <lemma tokID="t5">.</lemma>
 <lemma tokID="t6">sie</lemma>
 <lemma tokID="t7">schmecken</lemma>
 <lemma tokID="t8">er</lemma>
 <lemma tokID="t9">.</lemma>
</lemmas>
<parsing>
 <parse>
  <node ID="c1" cat="TOP">
   <node ID="c2" cat="S-TOP">
    <node ID="c3" cat="NP-SB">
     <node ID="c4" cat="PN-HD-Nom.Sg">
      <node ID="c5" cat="NE-HD-Nom.Sg">Peter</node>
     </node>
    </node>
    <node ID="c6" cat="VVFIN-HD">aß</node>
    <node ID="c7" cat="NP-OA">
     <node ID="c8" cat="ART-NK-Acc.Sg">eine</node>
     <node ID="c9" cat="NN-NK-Acc.Sg">Pizza</node>
    </node>
   </node>
   <node ID="c10" cat="\$.">.</node>
  </node>
 </parse>
 <parse>
  <node ID="c11" cat="TOP">
   <node ID="c12" cat="S-TOP">
    <node ID="c13" cat="NP-SB">
     <node ID="c14" cat="PPER-HD-Nom">Sie</node>
```

```
      </node>
      <node ID="c15" cat="VVFIN-HD">schmeckte</node>
      <node ID="c16" cat="NP-DA">
       <node ID="c17" cat="PPER-HD-Dat">ihm</node>
      </node>
     </node>
     <node ID="c18" cat="\$.">.</node>
    </node>
   </parse>
  </parsing>
  <depparsing>
   <parse>
    <dependency func="SUBJ" depID="t1" govID="t2"/>
    <dependency func="ROOT" depID="t2"/>
    <dependency func="SPEC" depID="t3" govID="t4"/>
    <dependency func="OBJ" depID="t4" govID="t2"/>
    <dependency func="SUBJ" depID="t6" govID="t7"/>
    <dependency func="ROOT" depID="t7"/>
    <dependency func="OBJ" depID="t8" govID="t7"/>
   </parse>
  </depparsing>
  <coreferences>
   <coreference>
    <referent constID="c3"/>
    <referent constID="c17"/>
   </coreference>
   <coreference>
    <referent constID="c7"/>
    <referent constID="c13"/>
   </coreference>
  </coreferences>
 </TextCorpus>
</CLARIN>
```

B RelaxNG Schema for CLARIN Documents

Contents of the main file *CLARIN.rnc*

```
element CLARIN {

  # The particular form of including external rnc-code below is chosen
  # in order to avoid conflicts which arise if one and the same pattern
  # name is used in two different external code files.
```

```
  # "lemmas" for instance is used in TextCorpus.rnc and Lexicon.rnc.

  grammar { include "MetaData.rnc" },
  (grammar { include "TextCorpus.rnc" } |
   grammar { include "Lexicon.rnc" })*
}
```

Contents of *TextCorpus.rnc*

```
start =
  element TextCorpus {

    attribute source { xsd:anyURI }?,
    attribute lang { xsd:language }?,
    attribute text { "yes" | "no" }?,
    attribute tokens { "yes" | "no" }?,
    attribute token_boundaries { "yes" | "no" }?,
    attribute sentences { "yes" | "no" }?,
    attribute sentence_boundaries { "yes" | "no" }?,
    attribute POStagging { text }?,
    attribute lemmatisation { "yes" | "no" }?,
    attribute coreferences { "yes" | "no" }?,
    attribute parsing { text }?,
    attribute depparsing { text }?,

    (\text | tokens | sentences | POStags | lemmas |
     parsing | depparsing | coreferences)+
  }

\text = element \text { text }

tokens =
  element tokens {
    element \token {
      attribute ID    { xsd:ID }?,
      (attribute start { xsd:integer },
       attribute end   { xsd:integer })?,
      text
    }+
  }

sentences =
  element sentences {
    element sentence {
```

```
      attribute ID    { xsd:ID }?,
      (attribute start { xsd:integer },
       attribute end   { xsd:integer })?,
      text
    }+
  }

POStags =
  element POStags {
    element tag {
      attribute tokID { xsd:IDREF },
      text
    }+
  }

lemmas =
  element lemmas {
    element lemma {
      attribute tokID { xsd:IDREF },
      text
    }+
  }

parsing =
  element parsing {
    element parse {
      node
    }+
  }

node =
  element node {
    attribute ID { xsd:ID }?,
    attribute cat { text },
    (node* | text)
  }

depparsing =
  element depparsing {
    element parse {
      element dependency {
        attribute func { text }?,
        attribute depID { xsd:IDREF },
        attribute govID { xsd:IDREF }?
```

```
      }+
    }+
  }

coreferences =
  element coreferences {
    element coreference { referent, referent+ }*
  }

referent =
  element referent {
    attribute constID { xsd:IDREF }
  }
```

Contents of *Lexicon.rnc* (A lot more needs to be added here in the future.)

```
start =
  element Lexicon {

    attribute lang { xsd:language }?,
    attribute frequencies { "yes" | "no" },

    lemmas, (frequencies | cooccurrences)*
  }

lemmas =
  element lemmas {
    element lemma {
      attribute ID { xsd:ID },
      text
    }+
  }

frequencies =
  element frequencies {
    element frequency {
      attribute lemmaID { xsd:IDREF },
      xsd:integer
    }+
  }

cooccurrences =
  element cooccurrences {
    attribute type { text }?,
```

```
    element cooccurrence {
      attribute func { text }?,
      attribute sig { xsd:float }?,
      term, term+
    }+
  }

term =
  element term {
    attribute lemID { xsd:IDREF }?,
    text?
  }
```

C Schematron Schema for CLARIN Documents

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<schema xmlns="http://www.ascc.net/xml/schematron" >

  <title>Schematron schemata for checking CLARIN documents</title>

  <pattern name="TextCorpus-text">
    <rule context="TextCorpus">
      <assert test="not(@text) or text">
        The 'TextCorpus' feature 'text' requires that a 'text' element
        is present!
      </assert>
    </rule>
  </pattern>

  <pattern name="TextCorpus-tokens">
    <rule context="token">
      <assert test="not(../../@tokens) or normalize-space(.) != ''">
        The 'TextCorpus' feature 'tokens' requires that each 'token'
        element contains text!
      </assert>
    </rule>
  </pattern>

  <pattern name="TextCorpus-token_boundaries">
    <rule context="TextCorpus">
      <assert test="not(@token_boundaries) or @text">
        The 'TextCorpus' feature 'token_boundaries' also requires the
```

```
      feature 'text'.
    </assert>
    <assert test="not(@token_boundaries) or tokens">
      The 'TextCorpus' feature 'token_boundaries' requires an element
      'tokens'.
    </assert>
  </rule>
  <rule context="token">
    <assert test="not(../../@token_boundaries) or @start">
      The 'TextCorpus' feature 'token_boundaries' requires that each
      token element has a 'start' attribute
    </assert>
    <assert test="not(../../@token_boundaries) or @end">
      The 'TextCorpus' feature 'token_boundaries' requires that each
      token element has an 'end' attribute
    </assert>
    <assert test="not(../../@token_boundaries) or @end >= @start">
      The value of the 'start' attribute must be smaller or equal to
      the value of the 'end' attribute!
    </assert>
    <assert test="not(../../@token_boundaries) or @start > 0">
      Negative values of the 'start' attribute are not allowed!
    </assert>
    <assert test="not(../../@token_boundaries) or
                  string-length(../../text) >= @end">
      The value of the 'end' attribute may not exceed the text length!
    </assert>
    <assert test="not(../../@token_boundaries) or
                  count(preceding-sibling::token)=0 or
                  @start > preceding-sibling::token[position()=1]/@end">
      The end position of a token may not overlap with the start
      position of the next token!
    </assert>
  </rule>
</pattern>

<pattern name="TextCorpus-POStagging">
  <rule context="TextCorpus">
    <assert test="not(@POStagging) or POStags">
      The 'TextCorpus' feature 'POStagging' requires an element 'POStags'.
    </assert>
  </rule>
  <rule context="POStags">
    <assert test="not(../@POStagging) or count(../tokens/token) = count(tag)">
```

```
        The 'TextCorpus' feature 'POStagging' requires an element 'tag' for
        each 'token'.
      </assert>
    </rule>
  </pattern>
</schema>
```