An Arranged Marriage: Integrating DKPro Core in the Language Analysis Portal

Milen Kouylekov[♣,♡], Emanuele Lapponi[♡], Stephan Oepen[♡], and Richard Eckart de Castilho[♠]
[♡]University of Oslo, Department of Informatics
[♣]University of Oslo, University Center for Information Technology
[♠]Technical University Darmstadt, Ubiquitous Knowledge Processing Lab
{milen|emanuel|oe}@ifi.uio.no
ekcart@ukp.informatik.tu-darmstadt.de

Abstract

In this paper we describe an ongoing effort to create an inter-operation framework that makes accessible the DKPro Core repository of Natural Language Processing (NLP) tools with the CLARINO Language Analysis Portal (LAP). The effort includes creating a mapping between the LAP Exchange Format and DKPro Core and integration of DKPro components in LAP.

1 Introduction: High-Level Goals

This paper describes an ongoing effort to create an inter-operation framework that makes accessible the DKPro Core repository of Natural Language Processing (NLP) components with the CLARINO Language Analysis Portal (LAP). Tight integration of the two projects will substantially enlarge the selection of NLP tools and their coverage across languages available to LAP users, and it will provide DKPro Core with an easy-to-use, in-browser user interface and broad availability as part of the CLARINO infrastructure. This work addresses interesting issues of metadata interpretation (in the description of input and output interfaces to DKPro Core component) as well as of interchange representations for the representation of diverse types of linguistic annotations.

The Language Analysis Portal (LAP; Lapponi et al. (2013)) is an ongoing initiative forming part of the CLARINO infrastructure project, the Norwegian branch of the pan-European CLARINO federation. The LAP objective is to provide an easily accessible web interface that eliminates technological barriers to entry for non-expert users. At the same time, by integrating a wide range of NLP tools and transparent access to high-performance computing (HPC), the portal enables execution of complex NLP workflows and ensures scalability to very large data sets. A core component of the current implementation is Galaxy (Giardine et al., 2005; Blankenberg et al., 2010; Goecks et al., 2010), a web-based workflow management system initially developed for data-intensive research in genomics and bioinformatics. In these fields, Galaxy portals allow biologists with no programming skills to access and configure processing tools, conduct experiments, and share both the results and the processing steps associated with them. By adapting Galaxy to the context of NLP, the vision of LAP is to ensure the same kind of access and ease of use of language technology tools for researchers from the humanities and the social sciences (and also, of course, for researchers within the field of NLP itself).

DKPro is a community of projects focusing on re-usable NLP software. At the heart of this project lies DKPro Core (Eckart de Castilho and Gurevych, 2014), which wraps a large array of off-the-shelf NLP tools and provides a common interface and API for the widely adopted UIMA framework (Ferrucci and Lally, 2004). Both LAP and DKPro Core achieve interoperability across tool-specific representations by converting inputs and outputs to their respective graph-based interchange formats. The effort described in this paper achieves compatibility between LAP and DKPro Core by developing a two-way format converter, allowing components from DKPro Core to interact with LAP tools and vice versa.

The reason why DKPro Core is a good choice for LAP lies not only in its broad coverage, but primarily in the fact that it comes as (automatically) downloadable software that can be easily deployed on demand and scaled out on the HPC infrastructure underlying LAP. There, it can process data locally and in a controlled environment. There is no need to transmit the data to remote (web) services and to monitor and manage the availability of such remote services.

The remainder of this abstract is structured as follows: Section 2 briefly describes commonalities and differences between the LAP and DKPro Core interchange formats for linguistic annotations; Section 3 describes the steps that were taken to tightly integrate DKPro Core components into LAP.

2 Mapping between the LAP Exchange Format and DKPro Core

A detailed comparison between the DKPro Core instantiation of the UIMA type system and the LAP eXchange Format was published by Eckart de Castilho et al. (2017). Following is a brief summary of relevant high-level characteristics for the present report.

DKPro Core Type System The DKPro Core Type System extends the type system that is built into the UIMA[1] framework (Ferrucci and Lally, 2004). It provides an annotation schema covering many layers of linguistic analysis, including segmentation, morphology, syntax, discourse, semantics, etc. Additionally, there are several types to encode metadata about the document being processed, about tagsets, etc. UIMA represents data using the Common Analysis System (CAS; **?**). The CAS consists of typed feature structures organized in a type system that supports single inheritance. There are various serialization formats for the CAS. The most prominent is based on the XML Metadata Interchange specification (XMI) (Stiller et al., 2002). However, there are also e.g. various binary serializations of the CAS with specific advantages, e.g. built-in compression for efficient network transfer.

LAP eXchange Format (LXF) The Linguistic Annotation Framework (LAF) is a graph-based model for representing multi-modal linguistic annotations that aims at providing full interoperability among annotation formats. Two fundamental principles underlying the LAF data model are (1) that annotation structure and annotation content should be separated and (2) that all annotation information should be explicitly represented (Ide and Suderman, 2014). Traditionally, neither principle is adhered to in off-the-shelf, standalone NLP tools such as tokenizers or part-of-speech (POS) taggers, where the output format is usually a direct reflection of the annotation content (e.g. normalized tokens or token–POS pairs) and significant information is only implicitly annotated. Importantly, LAF falls under the category of 'standoff' annotations, where the object of description and its annotations are not interspersed: In the case of text, the information represented in the annotation graphs references the media it annotates by means of so-called anchors to character indices. Closely following the ISO LAF guidelines, LXF represents annotations as a directed graph that references pieces of text; elements comprising the annotation graph and the base segmentation of the text are explicitly represented in LXF with a set of node, edge, and region elements. Example of a simple sentence represented in LXF is shown in Figure 1. The example contains a simple annotation of the text in the media attribute. The LXF contains one node that denotes that the text contains one Sentence. It also contains one region that gives the start and the end of the sentence according to the annotation tools in the anchors attribute.

Bi-Directional Conversion We use the LXF JSON serialization to send data as input to DKPro Core and import new annotations obtained by DKPro Core components as LXF-formatted output. To achieve this, we have contributed a dedicated LXF I/O module to DKPro Core. The module is maintained by the LAP team as part of the DKPro Core repository[2]. LAP annotations from the LAP Store are (i) exported to LXF upon invocation of a DKPro Core component, (ii) converted into UIMA CAS Feature Structures (FSes) defined by the DKPro Core type system and processed within DKPro Core; and (iii) communicated back to LAP, including new annotations, in LXF and ingested into the LAP Store. The new module essentially provides two converters, viz. a converter from LXF to DKPro Core types and an inverse converter, from DKPro Core types to LXF.

Converting LXF documents to DKPro Core type system FSes is done by converting the LXF graph nodes, edges, and regions[3] as required into *Sentence*, *Token*, *POS*, and *Dependency* FSes. The correspondence is as follows: 1) A node with type sentence is converted into a *Sentence* FS; 2) a node with

[1]When talking about UIMA, we refer to the Apache UIMA implementation; `http://uima.apache.org`.

[2]DKPro Core LXF module: `https://github.com/dkpro/dkpro-core/tree/master/dkpro-core-io-lxf-asl`

[3]LXF concepts are set in sans-serif and DKPro Core concepts in *italics*.

```
{"media": "CLARIN Annual Conference 2017 in Budapest, Hungary",
 "nodes": [ {
  "entity": "node",
  "origin": "tokenizer",
  "id": "tokenizer-n1@1",
  "index": 0,
  "rank": 0,
  "type": "sentence",
  "annotations": {
   "label": "CLARIN Annual Conference 2017 in Budapest, Hungary" } ],
 "edges": [],
 "regions": [ {
  "entity": "region",
  "origin": "tokenizer",
  "id": "tokenizer-r1@1",
  "index": 0,
  "anchors": [0, 55] } ] }
```

Figure 1: An example of LXF input to DKPro Core of a text with sentence boundary annotation.

type token is converted into a *Token* FS; 3) a node with type morphology is converted into a *POS* FS or a *Lemma* FS, depending on the value of the annotations inside the node; a node with type dependency is converted into *Dependency* FS.

For example the input from Figure 1 will be transformed into a document with the contents of the media attribute as its text. The document will have one *Sentence* FS with the values of the anchors attribute denoting the sentence boundaries.

The conversion between the LXF format and the UIMA CAS objects is straightforward. The LXF annotated file is loaded in Java objects with corresponding names Node, Region and Edge. The standoff LXF annotation is directly converted into the standoff annotation of the DKPro Core type system by creating the corresponding FS in the UIMA CAS. The Node objects are converted into annotations and the Region objects provide the corresponding offsets. As for the conversion of LXF node annotations into DKPro Core typed FSes, this process requires an interpretation of the LXF annotations.

The process involves recognizing annotations with specific names as nodes of certain types, and converting them into the corresponding DKPro Core types. For example, a node with type morphology and annotation pos is converted into a *POS* FS, and the value of this annotation is set as the *posValue* of the object. This newly created object is then associated with the token object corresponding to the LXF token node with an incoming edge from the original POS node. The resulting mapping is as follows:

- { 'type': 'sentence', 'annotations':{'label': "Sentence text"}} → Sentence()
- { 'type': 'token', 'annotations':{'label': "token text"}}
  → Token(text=value('label'))
- { 'type': 'morphology', 'annotations':{'pos': "pos value", 'lemma' : 'lemma value'}}
  → Token(posValue=value('pos'), lemma=value('lemma'))
- { 'type': 'dependency', 'annotations':{'label': "dependency relation label"}}
  → Dependency(dependencyType=value('label'))

The region linked to the node of type sentence in LXF will initialize the sentence start and end offset in the UIMA CAS object. Correspondingly, the region linked to the node of type token will initialize the token offsets. For the nodes of type dependency, LXF edges are used to determine the Governor token and the Dependent token. it is important to note that LXF is generic, thus the tools in LAP can create annotations with different names and semantics. In order to ensure that possibly open-ended output is correctly interpreted, only annotations matching one of the mappings above are allowed. If a tool in LAP

```
{
 "nodes": [ {
  "entity": "node",
  "origin": "dkpro",
  "id": "dkpro-n1@1",
  "index": 0,
  "rank": 0,
  "type": "token",
  "annotations": {
    "label": "CLARIN" } }, ...],
 "edges": [{
   "origin": "dkpro",
    "index": 0,
    "from": "dkpro-n1@1",
    "rank": 0,
    "entity": "edge",
    "to": "tokenizer-n1@1",}, ...],
 "regions": [ {
  "entity": "region",
  "origin": "dkpro",
  "id": "dkpro-r1@1",
  "index": 0,
  "anchors": [0, 6 ]}, .... ] }
```

Figure 2: An example of LXF output of DKPro Core of the fist token in the example sentence with the token boundary annotation.

produces an incompatible annotation, its output will not be loaded correctly in DKPro Core UIMA CAS object.

Converting a document annotated using DKPro Core into LXF involves creating corresponding LXF nodes, edges and regions for each of the objects in the DKPro type system. For example, if the LXF document shown in Figure 1 is annotated with a tokenizer in DKPro Core, for each *Token* UIMA CAS object created by the tokenizer a node of type token will be added to the LXF graph. Each of these tokens will have a corresponding edge to the sentence node and a corresponding region denoting its position in the text. Similarly, if the token is tagged by a part-of-speech tagger, for each POS tag a node with type morphology will be created in the LXF graph and an edge will connect the new node with the corresponding token node.

The output document shown inf Figure 2 contains only the new annotations created by a DKPro Core tool, depending on the parameterization of the LXF Writer in DKPro Core. Here, the first token (*"CLARIN"*) created by a DKPro tool is converted in LXF json format. One node is created with the annotation label containing the token text. The region corresponding to this token stores the offset of the token in the text. An edge is added to denote that the token node created belongs to the sentence node in the input annotation.

3 DKPro Core Components in LAP

To instantiate this new bi-directional bridge between LAP and DKPro Core, we first 'manually' wrapped two DKPro Core components: a (sentence) splitter-tokenizer and a tagger-parser (both from the CoreNLP eco-system). These components are currently available for experimentation in the LAP production instance. The procedure is done in the following steps: 1) the wrapper retrieves annotations from the internal LAP storage to create LXF input for DKPro Core; 2) the LXF reader is called to create the appropriate UIMA CAS internal representation of the input LXF; 3) the CoreNLP segmenter and CoreNLP parser modules of DKPro Core are invoked on the converted input; 4) the output of the annotation components is converted back to LXF; and 5) the output LXF is obtained from DKPro Core and ingested into the LAP annotation storage.

Next, a generalized wrapper script was created to execute this process from within LAP. The script can be configured using the following parameters: 1) DKPro Core component name; 2) a set of annotations required as input for the component (for example, sentence and token boundary annotations are obligatory before invoking a tagger). The LXF output of DKPro Core will not contain the annotations given to it as an input, allowing to directly import the document into the internal LAF storage as a new annotation.

The addition of the DKPro Core components to LAP substantially increases the number of tools available in LAP, and provides a convenient browser-based access to the DKPro Core infrastructure.

To avoid manually wrapping the considerable number of DKPro Core components, we make use of DKPro Meta[4], a companion project to DKPro Core. DKPro Meta collects and aggregates metadata about DKPro Core, including the list of available components, which models are available for these components, which languages are supported, and much more. DKPro Meta is used to drive the automatic generation of the reference documentation of DKPro Core. However, with a custom set of templates, it can also be used as a code generator to create LAP-compatible wrapper scripts and Galaxy tool descriptions for DKPro Core components.

We have chosen five types of components that are appropriate for integration in LAP: 1) splitter–tokenizer (combinations of sentence splitting and/or tokenization); 2) tagger (token-wise part of speech tagging); 3) lemmatizer (token-wise lemmatization); 4) tagger–parser (syntactic parsing with or without joint tagging). Instead of exposing each DKPro Core component instantiating the categories as separate tools in LAP, we decided to expose each category as a single tool, e.g. *DKPro Core POS Tagger*. The actual DKPro Core component wrapping a specific third-party tool is then chosen as a parameter in the LAP user interface. This approach avoids overloading the tool menu in Galaxy with a large number of items.

The splitter–tokenizer and tagger–parser components using CoreNLP tools in DKPro Core are already integrated in the experimental section of the language analysis portal. Integration of more DKPro Core components from the above categories into the production instance of LAP is reaching completion in the Fall of 2017 and will add dozens of new tools and expand the coverage of the portal beyond English and Norwegian.

Conclusions

In this article we presented our ongoing work on integrating the DKPro Core repository of language processing tools into the CLARINO Language Analysis Portal. The integration is straightforward and allows us to integrate many tools that form part of DKPro Core with the use of a single wrapper, rather than creating different wrappers for each tool. This integration will storngly increase coverage of the Language Analysis Portal of both tools and supported languages. The integration also demonstrated that our LXF format is flexible enough to be used as an interchange format with DKPro. Taking into account a simple set of rules for interpretation, a module for input conversion was created in DKPro. The output of DKPro was seamlessly exported back in LXF and imported into the LAP internal database.

Acknowledgments

This work is part of the Norwegian CLARINO initiative, the national branch of the pan-European Common Language Resources and Technology Infrastructure (CLARIN). For further background and access to the CLARINO Language Analysis Portal, please see `http://www.mn.uio.no/ifi/english/research/projects/clarino`.

Furthermore, the contributions of the fourth author were funded in parts from the European Union's Horizon 2020 research and innovation programme (H2020-EINFRA-2014-2) under grant agreement No. 654021. It reflects only the author's views and the EU is not liable for any use that may be made of the information contained therein. It was further supported in parts by the German Federal Ministry of Education and Research (BMBF) under the promotional reference 01UG1416B (CEDIFOR).

[4]DKPro Meta: `https://github.com/dkpro/dkpro-meta`

References

Daniel Blankenberg, Gregory Von Kuster, Nathaniel Coraor, Guruprasad Ananda, Ross Lazarus, Mary Mangan, Anton Nekrutenko, and James Taylor. 2010. Galaxy. A web-based genome analysis tool for experimentalists. *Current Protocols in Molecular Biology*, page 19.10.1 – 21, January.

Richard Eckart de Castilho and Iryna Gurevych. 2014. A broad-coverage collection of portable nlp components for building shareable analysis pipelines. In *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT*, pages 1–11, Dublin, Ireland, August. Association for Computational Linguistics and Dublin City University.

Richard Eckart de Castilho, Nancy Ide, Emanuele Lapponi, Stephan Oepen, Keith Suderman, Erik Velldal, and Marc Verhagen. 2017. Representation and interchange of linguistic annotation. an in-depth, side-by-side comparison of three designs. In *Proceedings of the 11th Linguistic Annotation Workshop*, pages 67–75, Valencia, Spain, April. Association for Computational Linguistics.

David Ferrucci and Adam Lally. 2004. UIMA. An architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327 – 348, September.

Belinda Giardine, Cathy Riemer, Ross C. Hardison, Richard Burhans, Laura Elnitski, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, James Taylor, Webb Miller, W. James Kent, and Anton Nekrutenko. 2005. Galaxy. A platform for interactive large-scale genome analysis. *Genome Research*, 15(10):1451 – 5, October.

Jeremy Goecks, Anton Nekrutenko, James Taylor, and The Galaxy Team. 2010. Galaxy. A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8:R86), August.

Nancy Ide and Keith Suderman. 2014. The Linguistic Annotation Framework. A standard for annotation interchange and merging. *Language Resources and Evaluation*, 48(3):395 – 418.

Emanuele Lapponi, Erik Velldal, Nikolay A. Vazov, and Stephan Oepen. 2013. Towards large-scale language analysis in the cloud. In *Proceedings of the 19th Nordic Conference of Computational Linguistics: Workshop on Nordic Language Research Infrastructure*, page 1 – 10, Oslo, Norway.

Burkhard Stiller, Thomas Bocek, Fabio Hecht, Guilherme Machado, Peter Racz, and Martin Waldburger. 2002. OMG XML metadata interchange (XMI) specification. Technical report, IOMG.