Representation and Interchange of Linguistic Annotation
An In-Depth, Side-by-Side Comparison of Three Designs

Richard Eckart de Castilho♣, Nancy Ide♠, Emanuele Lapponi♡, Stephan Oepen♡, Keith Suderman♠, Erik Velldal♡, and Marc Verhagen◊

♣ Technische Universität Darmstadt, Department of Computer Science
♠ Vassar College, Department of Computer Science
♡ University of Oslo, Department of Informatics
◊ Brandeis University, Linguistics and Computational Linguistics

Abstract

For decades, most self-respecting linguistic engineering initiatives have designed and implemented *custom representations* for various layers of, for example, morphological, syntactic, and semantic analysis. Despite occasional efforts at harmonization or even standardization, our field today is blessed with a multitude of ways of encoding and exchanging linguistic annotations of these types, both at the levels of 'abstract syntax', naming choices, and of course file formats. To a large degree, it is possible to work within and across design plurality by *conversion*, and often there may be good reasons for divergent design reflecting differences in use. However, it is likely that some abstract commonalities across choices of representation are obscured by more superficial differences, and conversely there is no obvious procedure to tease apart what actually constitute contentful vs. mere technical divergences. In this study, we seek to conceptually align three representations for common types of morpho-syntactic analysis, pinpoint what in our view constitute contentful differences, and reflect on the underlying principles and specific requirements that led to individual choices. We expect that a more in-depth understanding of these choices across designs may lead to increased harmonization, or at least to more informed design of future representations.

1 Background & Goals

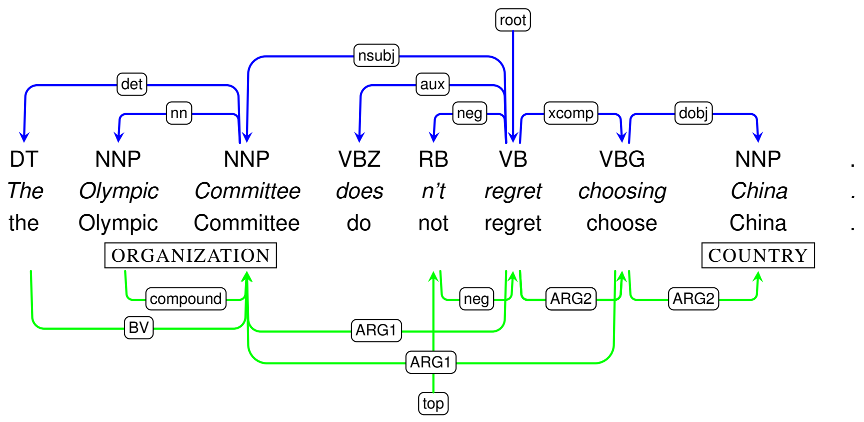
This study is grounded in an informal collaboration among three frameworks for 'basic' natural language processing, where *workflows* can combine the outputs of processing tools from different developer communities (i.e. software repositories), for example a sentence splitter, tokenizer, lemmatizer, tagger, and parser—for morpho-syntactic analysis of running text. In large part owing to divergences in input and output representations for such tools, it tends to be difficult to connect tools from different sources: Lacking interface standardization, thus, severely limits *interoperability*.

The frameworks surveyed in this work address interoperability by means of a common representation—a uniform framework-internal convention—with mappings from tool-specific input and output formats. Specifically, we will take an in-depth look at how the results of morpho-syntactic analysis are represented in (a) the DKPro Core component collection[1] (Eckart de Castilho and Gurevych, 2014), (b) the Language Analysis Portal[2] (LAP; Lapponi et al. (2014)), and (c) the Language Application (LAPPS) Grid[3] (Ide et al., 2014a). These three systems all share the common goal of facilitating the creation of complex NLP workflows, allowing users to combine tools that would otherwise need input and output format conversion in order to be made compatible. While the programmatic interface of DKPro Core targets more technically inclined users, LAP and LAPPS are realized as web applications with a point-and-click graphical interface. All three have been under active development for the past several years and have—in contemporaneous, parallel work—designed and implemented framework-specific representations. These designs are rooted in related but interestingly different traditions; hence, our side-by-side discussion of these particular frameworks provides a good initial sample of observable commonalities and divergences.[4]

2 Terminological Definitions

A number of closely interrelated concepts apply to the discussion of design choices in the repre-

[1] `https://dkpro.github.io/dkpro-core`
[2] `https://lap.clarino.uio.no`
[3] `https://www.lappsgrid.org`
[4] There are, of course, additional designs and workflow frameworks that we would ultimately hope to include in this comparison, as for example the representations used by CONCRETE, WebLicht, and FoLiA (Ferraro et al., 2014; Heid et al., 2010; van Gompel and Reynaert, 2013), to name just a few. However, some of these frameworks are at least abstractly very similar to representatives in our current sample, and also for reasons of space we need to restrict this in-depth comparison to a relatively small selection.



root
nsubj
det
aux
nn
neg
xcomp
dobj
DT NNP NNP VBZ RB VB VBG NNP .
The Olympic Committee does n't regret choosing China .
the Olympic Committee do not regret choose China .
ORGANIZATION
COUNTRY
compound
neg
ARG2
ARG2
BV
ARG1
ARG1
top

Figure 1: Running example, in 'conventional' visualization, with five layers of annotation: syntactic dependencies and parts of speech, above; and lemmata, named entities, and semantic dependencies, below.

sentations for linguistic annotations. Albeit to some degree intuitive, there is substantial terminological variation and vagueness, which in turn reflects some of the differences in overall annotation scheme design across projects and systems. Therefore, with due acknowledgement that no single, definitive view exists we provide informal definitions of relevant terms as used in the sections that follow in order to ground our discussion.

Annotations For the purposes of the current exercise we focus on annotations of language data in textual form, and exclude consideration of other media such as speech signals, images, and video. An *annotation* associates linguistic information such as morpho-syntactic tags, syntactic roles, and a wide range of semantic information with one or more spans in a text. Low-level annotations typically segment an entire text into contiguous spans that serve as the base units of analysis; in text, these units are typically sentences and tokens. The association of an annotation to spans may be direct or indirect, as an annotation can itself be treated as an object to which other (higher level) annotations may be applied.

Vocabulary The *vocabulary* provides an inventory of semantic entities (concepts) that form the building blocks of the annotations and the relations (links) that may exist between them (e.g. constituent or dependency relations, coreference, and others). The CLARIN Data Concept Registry[5] (formerly ISOcat) is an example of a vocabulary for linguistic annotations.

[5]`https://openskos.meertens.knaw.nl/ccr/browser/`

Schema A *schema* provides an abstract specification (as opposed to a concrete realization) of the structure of annotation information, by identifying the allowable relations among entities from the vocabulary that may be expressed in an annotation. A schema is often expressed using a diagrammatic representation such as a UML diagram or entity-relationship model, in which entities label nodes in the diagram and relations label the edges between them. Note that the vocabulary and the schema that uses it are often defined together, thus blurring the distinction between them, as for example, in the LAPPS Web Service Exchange Vocabulary (see Section 3) or any UIMA CAS type system.

Serialization A serialization of the annotations is used for storage and exchange. Annotations following a given schema can be serialized in a variety of formats such as (basic) XML, database (column-based) formats, compact binary formats, JSON, ISO LAF/GrAF (Ide and Suderman, 2014; ISO, 2012), etc.

3 A Simple Example

In the following sections, we will walk through a simple English example with multiple layers of linguistic annotations. Figure 1 shows a rendering of our running example in a compact, graphical visualization commonly used in academic writing. However, even for this simple sentence, we will point out some hidden complexity and information left implicit at this informal level of representation.

For example, there are mutual dependencies among the various layers of annotation: parts of speech and lemmatization both encode aspects of

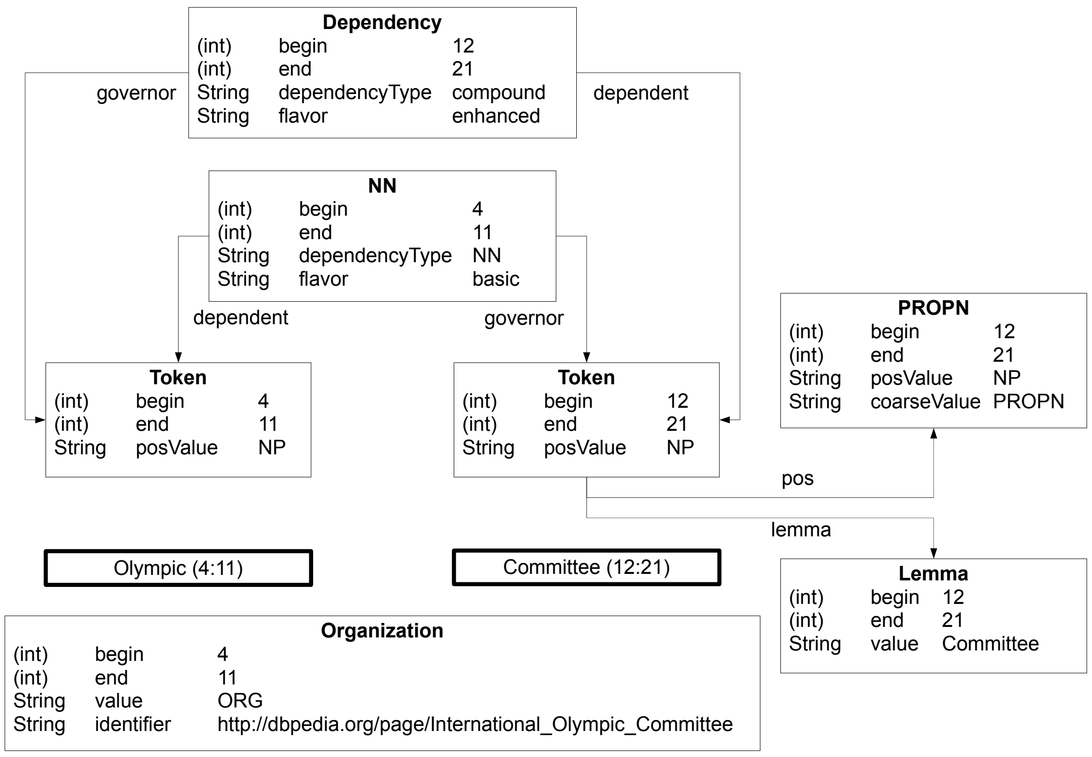


Figure 2: DKPro Core Zoom

token-level morphological analysis; syntactic dependencies, in turn, are plausibly interpreted as building on top of the morphological information; finally, also the semantic dependencies, will typically be based on some or maybe all layers of morpho-syntactic analysis. In practice, on the other hand, various layers of annotation are often computed by separate tools, which may or may not take into account information from 'lower' analysis layers. In this respect, the visualization in Figure 1 gives the impression of a single, 'holistic' representation, even though it need not always hold that all analysis layers have been computed to be mutually consistent with each other. In Section 4 below, we will observe that a desire to make explicit the *provenance* of separate annotation layers can provide an important design constraint.

DKPro Core Type System The DKPro Core Type System extends the type system that is built into the UIMA[6] framework (Ferrucci and Lally, 2004). It provides types for many layers of linguistic analysis, such as segmentation, morphology, syntax, discourse, semantics, etc. Additionally, there are several types that carry metadata about the document being processed, about tagsets, etc.

UIMA represents data using the Common Analysis System (CAS) (Götz and Suhre, 2004). 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) (OMG, 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.

The built-in types of UIMA are basic ones, such as *TOP* (a generic feature structure), *Annotation* (a feature structure anchored on text via start/end offsets), or *SofA* (short for 'subject of analysis', the signal that annotations are anchored on). A single CAS can accommodate multiple *SofA*s which is useful in many cases: holding multiple translations of a document; holding a markup and a plaintext version; holding an audio signal and its transcription; etc. *Annotation* and all types inheriting from it are always anchored on a single *SofA*, but they may refer to annotations anchored on a different *SofA*, e.g. to model word alignment between translated texts. The CAS also allows for feature structures inheriting from *TOP* that are not anchored to a particular *SofA*.

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

Figure 2 shows a zoom on the sub-string *Olympic Committee* from Figure 1. All of the shown annotations are anchored on the text using offsets. The *Token*, PoS (*PROPN*), and *Lemma* annotations are anchored each on a single word. The named entity (*Organization*) annotation spans two words. The *Dependency* relation annotations anchored by convention on the span of the dependent *Token*. Syntactic and semantic dependencies are distinguished via the *flavor* feature. All annotations (except *Token*) have a feature which contains the original output(s) of the annotation tool (*value, posValue, coarseValue, dependencyType*). The possible values for these original outputs are not specified in the DKPro Core type system. However, for the convenience of the user, DKPro Core supports so-called *elevated types* which are part of the type system and function as universal tags. Using mappings, the elevated type is derived from the original tool output. The Penn Treebank tag NNP, for example. is mapped to the type PROPN. For example, for PoS tags DKPro Core uses the Universal Dependency PoS categories.

Lap eXchange Format (LXF) Closely following the ISO LAF guidelines (Ide and Suderman, 2014), 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. Regions describe text segmentation in terms of character offsets, while nodes contain (sets of) annotations (and optionally direct *links* to regions). Edges record relations between nodes and are optionally annotated with (non-linguistic) structural information. Nodes in LXF are typed, ranging for example over *sentence*, *token*, *morphology*, *chunk*, or *dependency* types. There are some technical properties common to all nodes (e.g. a unique identifier, sequential index, as well as its *rank* and *receipt*, as discussed below), and each node further provides a feature structure with linguistic annotations, where the particular range of features is determined by the node type.

Consider the example of Figure 1. Assuming sentence segmentation prior to word tokenization, the corresponding LXF graph comprises ten regions, one for the full sentence and one for each of the tokens. Figure 3 pictures the LXF version of Figure 2 described in the previous section. For sentence segmentation, LXF includes one node of type *sentence*, which contains links (represented as dashed edges in Figure 3) to the corresponding regions; similarly, *token*-typed nodes also directly link to their respective region, effectively treating sentence segmentation and word tokenization equally. If these two annotation types are obtained sequentially as part of an annotation *workflow* (i.e. the word tokenizer segments one sentence at a time), the LXF graph includes one directed edge from each token node to its sentence node, thus ensuring that the *provenance* of the annotation is explicitly modeled in the output representation.

Moving upwards to parts of speech, LXF for the example sentence includes one node (of type *morphology*) per PoS, paired with one edge pointing to the token node it annotates. Similarly to the sentence–token relationship, separate *morphology* nodes report the result of lemmatization, with direct edges to PoS nodes when lemmatization requires tagged input (or to *token* nodes otherwise). In general, running a new tool results in new LXF (nodes and) edges linking incoming nodes to the highest (i.e. topologically farthest from segmentation) annotation consumed. This holds true also for the nodes of type *dependency* in Figure 3; here each dependency arc from Figure 1 is 'reified' as a full LXF node, with an incoming and outgoing edge each recording the directionality of the head–dependent relation. The named entity, in turn, is represented as a node of type *chunk*, with edges pointing to nodes for PoS tags, reflecting that the named entity recognizer operated on tokenized and tagged input.

LXF graph elements (including the annotated media and regions) are in principle serialization-agnostic, and currently implemented in LAP as a multitude of individual, atomic records in a NoSQL database. A specific annotation (sub-)graph, i.e. a collection of interconnected nodes and edges, in this approach is identified by a so-called *receipt*, essentially a mechanism for group formation. Each step in a LAP workflow consumes one or more receipts as input and returns a new receipt comprising additional annotations. Thus, each receipt uniquely identifies the set of annotations contributed by one tool, as reflected in the receipt properties on the nodes of Figure 3. LAP imposes a strict principle of monotonicity, meaning that existing annotations are never modified by later processing, but rather each tool adds its own, new layer of annotations (which could in principle 'overlay' or 'shadow'

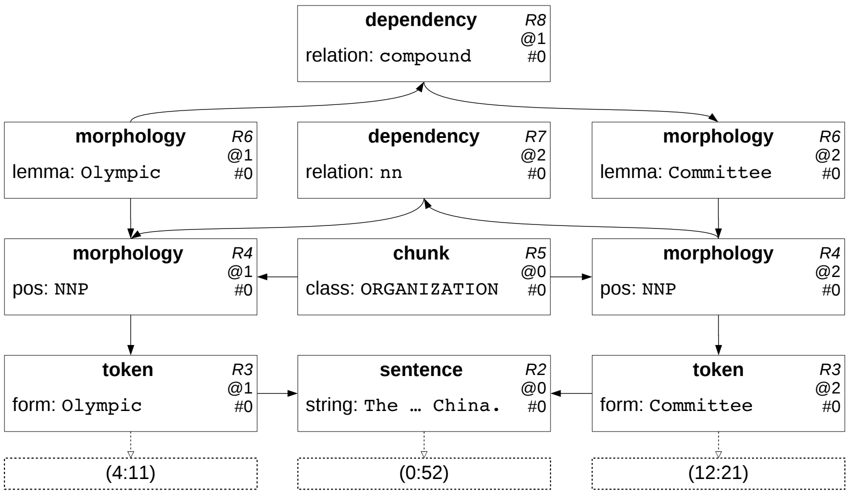


Figure 3: Excerpt from the LXF graph for the running example in Figure 1, zooming in on *Olympic Committee*. Segmentation regions are shown as dashed boxes, with character offsets for the two tokens and the full sentence, respectively. Nodes display their type (e.g. *morphology*), part of the feature structure containing the linguistic annotation (e.g. the part of speech), and their receipt, index, and rank properties ('R4', '@1' and '#0', respectively).

information from other layers). Therefore, for example, parallel runs of the same (type of) tool can output graph elements that co-exist in the same LXF graph but can be addressed each by their own receipt (see Section 4 below).

LAPPS Interchange Format (LIF) The LAPPS Grid exchanges annotations across web services using LIF (Verhagen et al., 2016), an instantiation of JSON-LD (JavaScript Object Notation for Linked Data), a format for transporting Linked Data using JSON, a lightweight, text-based, language-independent data interchange format for the portable representation of structured data. Because it is based on the W3C Resource Definition Framework (RDF), JSON-LD is trivially mappable to and from other graph-based formats such as ISO LAF and UIMA CAS, as well as a growing number of formats implementing the same data model. JSON-LD extends JSON by enabling references to annotation categories and definitions in semantic-web vocabularies and ontologies, or any suitably defined concept identified by a URI. This allows for referencing linguistic terms in annotations and their definitions at a readily accessible canonical web location, and helps ensure consistent term usage across projects and applications. For this purpose, the LAPPS Grid project provides a Web Service Exchange Vocabulary (WSEV; Ide et al. (2014b)), which defines a schema comprising an inventory of web-addressable entities and relations.[7]

Figure 4 shows the LIF equivalent of Figures 2 and 3 in the previous sections. Annotations in LIF are organized into *views*, each of which provides information about the annotations types it contains and what tool created the annotations. Views are similar to annotation 'layers' or 'tasks' as defined by several mainstream annotation tools and frameworks. For the full example in Figure 1, a view could be created for each annotation type in the order it was produced, yielding six consecutive views containing sentence boundaries, tokens, parts of speech and lemmas, named entities, syntactic dependencies, and semantic dependencies.[8] In Figure 4, a slightly simplified graph is shown with only three views and where token and part of speech information is bundled in one view and where lemmas and semantic relations are ignored. A view

[7]The WSEV links terms in the inventory to equivalent or similar terms defined elsewhere on the web.

[8]The last three views could be in a different order, depending on the sequence in which the corresponding tools were applied.

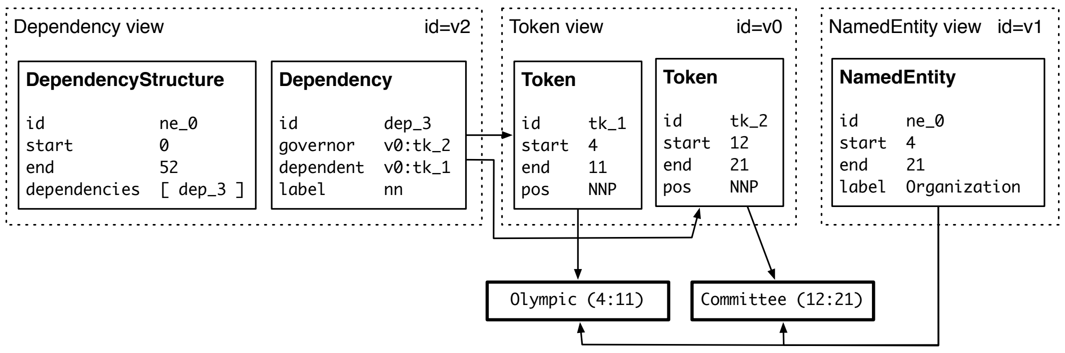


Figure 4: Excerpt from the LIF graph for the running example in Figure 1, zooming in on *Olympic Committee*. Views are shown as dashed boxes and annotations as regular boxes with annotation types in bold and with other attributes as appropriate for the type. Arrows follow references to other annotations or to the source data.

is typically created by one processing component and will often contain all information added by that component. All annotations are in standoff form; an annotation may therefore reference a span (region) in the primary data, using character offsets, or it may refer to annotations in another view by providing the relevant ID or IDs. In the example, a named entity annotation in the Named Entity view refers to character offsets in the primary data and the dependency annotation in the Dependency view refers to tokens in the Token view, using Token view ID and the annotation IDs as defined in the Token view.[9]

Preliminary Observations Each of the three representations has been created based on a different background and with a different focus. For example, LIF is coupled with JSON-LD as its serialization format and uses *views* to carry along the full analysis history of a document including per-view provenance data. LXF uses explicit relations between annotations to model provenance and has strong support for multiple concurrent annotations and for managing annotation data persisted in a database. The DKPro Core is optimized for ease of use and processing efficiency within analysis workflows and has rather limited support for concurrent annotations, provenance, and stable IDs.

Some technical differences between the three designs actually remain hidden in the abbreviated, diagrammatic representations of Figures 2 to 4. Abstractly, all three are directed graphs, but the relations between nodes in DKPro Core and LIF are established by having the identifier or reference of the target node as a feature value on a source node, whereas LXF (in a faithful rendering of the ISO LAF standard) actually realizes each edge as a separate, structured object. In principle, separate edge objects afford added flexibility in that they could bear annotations of their own—for example, it would be possible to represent a binary syntactic or semantic dependency as just one edge (instead of reifying the dependency as a separate node, connected to other nodes by two additional edges). However, Ide and Suderman (2014) recommend to restrict edge annotations to non-linguistic information, and LXF in its current development status at least heeds that advice. Hence, the DKPro Core and LIF representations are arguably more compact (in the number of objects involved).

A broad view of the three approaches shows that at what we may regard as the ‘micro-level’, that is, the representation of individual annotations, differences are irrelevant in terms of the *schema* applied, which are trivially mappable based on a common underlying (graph-based) model. At a higher level, however, different goals have led to divergences in the content and organization of the information that is sent from one tool to another in a workflow chain. In the following section, we consider these differences.

4 Pushing a Little Farther

While our above side-by-side discussion of ‘basic’ layers of morpho-syntactic annotations may seem to highlight more abstract similarity than divergence, in the following we will discuss a few more

[9]Note that multiple Token views can co-exist in the annotation set.

intricate aspects of specific annotation design. We expect that further study of such 'corner cases' may shed more light on inherent degrees of flexibility in a particular design, as well as on its scalability in annotation complexity.

Media–Tokenization Mismatches Tokenizers may apply transformations to the original input text that introduce character offset mismatches with the normalized output. For example, some Penn Treebank–compliant tokenizers normalize different conventions for quotation marks (which may be rendered as straight 'typewriter' quotes or in multi-character LaTeX-style encodings, e.g. `"` or ` `` `) into opening (left) and closing (right) Unicode glyphs (Dridan and Oepen, 2012). To make such normalization accessible to downstream processing, it is insufficient to represent tokens as only a region (sub-string) of the underlying linguistic signal.

In LXF, the string output of tokenizers is recorded in the annotations encapsulated with each *token* node, which is in turn linked to a region recording its character offsets in the original media. LIF (which is largely inspired by ISO LAF, much like LXF) also records the token string and its character offsets in the original medium. LIF supports this via the *word* property on tokens. DKPro Core has also recently started introducing a *TokenForm* annotation optionally attached to *Token* feature structures to support this.

Tokenizers may also return more than one token for the same region. Consider the Italian word *del*, which combines the preposition *di* and the definite article *il*. With both tokens anchored to the same character offsets, systems require more than reasoning over sub-string character spans to represent the linear order of tokens. LXF and LIF encode the ordering of annotations in an index property on nodes, trivializing this kind of annotation. DKPro Core presently does not support this.

Alternative Annotations and Ambiguity While relatively uncommon in the manual construction of annotated corpora, it may be desirable in a complex workflow to allow multiple annotation layers of the same type, or to record in the annotation graph more than the one-best hypothesis from a particular tool. Annotating text with different segmenters, for example, may result in diverging base units, effectively yielding parallel sets of segments. In our running example, the contraction *don't* is conventionally tokenized as ⟨`do`,`n't`⟩, but a linguistically less informed tokenization regime might also lead to the three-token sequence ⟨`don`,`'`,`t`⟩ or just the full contraction as a single token.

In LXF, diverging segmentations originating from different annotators co-exist in the same annotation graph. The same is true for LIF, where the output of each tokenizer (if more than one is applied) exists in its own view with unique IDs on each token, which can be referenced by annotations in views added later. Correspondingly, alternative annotations (i.e. annotations of the same type produced by different tools) are represented with their own set of nodes and edges in LXF and their own views in LIF. The DKPro Core type system does not link tokens explicitly to the sentence but relies on span offsets to infer the relation. Hence, it is not possible to represent multiple segmentations on a single *SofA*. However, it is possible to have multiple *SofA*s with the same text and different segmentations within a single CAS.

A set of alternative annotations may also be produced by a single tool, for instance in the form of an n-best list of annotations with different confidence scores. In LXF, this kind of ambiguous analyses translates to a set of graph elements sharing the same receipt identifier, with increasing values of the *rank* property for each alternative interpretation. Again, the DKPro Core type system largely relies on span offsets to relate annotations to tokens (e.g. named entities). Some layers, such as dependency relations also point directly to tokens. However, it is still not possible to maintain multiple sets of dependency relations in DKPro Core because each relation exists on its own and there is presently nothing that ties them together. The views in LIF are the output produced by any single run of a given tool over the data; therefore, in this case all the variants would be contained in a single view, and the alternatives would appear in a list of values associated with the corresponding feature (e.g. a list of PoS–confidence score pairs). Additionally, LIF provides a *DependencyStructure* which can bind multiple dependency relations together and thus supports multiple parallel dependency structures even within a single LIF view.

Parallel Annotation At times it is necessary to have multiple versions of a text or multiple parallel texts during processing, e.g. when correcting mistakes, removing markup, or aligning translations. DKPro Core inherits from the UIMA CAS the ability to maintain multiple *SofA*s in parallel. This features of UIMA is for example used in the DKPro Core normalization framework where *SofaChangeAnnotations* can be created on one view, stating that text should be inserted, removed, or replaced. These annotations can then be *applied* to the text using a dedicated component which creates a new *SofA* that contains the modified text. Further processing can then happen on the modified text without any need for the DKPro Core type system or DKPro Core components to be aware of the fact that they operate on a derived text. The alignment information between the original text and the derived text is maintained such that the annotations created on the derived text can be transferred back to the original text.

The LXF and LIF designs support multiple layers or views with annotations, but both assume a single base text. In these frameworks, text-level edits or normalizations would have to be represented as 'overlay' annotations, largely analogous to the discussion of token-level normalization above.

Provenance Metadata describing the software used to produce annotations, as well as the rules and/or annotation scheme—e.g. tokenization rules, part-of-speech tagset—may be included with the annotation output. This information can be used to validate the compatibility of input/output requirements for tool sequences in a pipeline or workflow.

LIF provides all of this information in metadata appearing at the beginning of each view, consisting of URI pointing to the producing software, tagset, or scheme used, and accompanying rules for identifying the annotation objects (where applicable).

The LXF principle of monotonicity in accumulating annotation layers is key to its approach to provenance. For our running example in Section 3 above, we assume that PoS tagging and lemmatization were applied as separate steps; hence, there are separate nodes for these (all of type *morphology*) and two distinct receipts. Conversely, if a combined tagger–lemmatizer had been used, its output would be recorded as a single layer of *morphology* nodes—yielding a different (albeit equivalent in linguistic content) graph structure.

The provenance support in DKPro Core is presently rather limited but also distinctly different from LXF or LIF. Presently, for a given type of annotation, e.g. PoS tags, the name of one creator component can be stored. This assumes that every type of annotation is produced by at most by one component. Additionally, whenever possible, DKPro Core extracts tagsets from the models provided with taggers, parsers, and similar components and stores these along with the model language, version, and name in a *TagsetDescription*. This even lists tags not output by the component.

5 Conclusions & Outlook

We have surveyed the differences and commonalities among three workflow analysis systems in order to move toward identifying the needs to achieve greater interoperability among workflow systems for NLP. This preliminary analysis shows that while some basic elements have common models and are therefore easily usable by other systems, the handling of *alternative* annotations and representation of provenance are among the primary differences in approach. This suggests that future work aimed at interoperability needs to address this level of representation, as we attempt to move toward means to represent linguistically annotated data and achieve universal interoperability and accessibility.

In ongoing work, we will seek to overcome remaining limitations through (a) incremental refinement (working across the developer communities involved) that seeks to eliminate unnecessary, superficial differences (e.g. in vocabulary naming choices) and (b) further exploring the relationships between distinct designs via the implementation of a bidirectional converter suite. Information-preserving round-trip conversion, on this view, would be a strong indicator of abstract design equivalence, whereas conversion errors or information loss in round-trip conversion might either point to contentful divergences or room for improvement in the converter suite.

Acknowledgments

This work was 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) and by the U.S. National Science Foundation grants NSF-ACI 1147944 and NSF-ACI 1147912. LAP development is supported by the Norwegian Research Council through the national CLARINO initiative, as well as by the Norwegian national computing and storage e-infrastructures, and the Department of Informatics at the University of Oslo. We are grateful to everyone involved, in particular all taxpayers.

References

Rebecca Dridan and Stephan Oepen. 2012. Tokenization. Returning to a long solved problem. A survey, contrastive experiment, recommendations, and toolkit. In *Proceedings of the 50th Meeting of the Association for Computational Linguistics*, page 378 – 382, Jeju, Republic of Korea, July.

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*, page 1 – 11, Dublin, Ireland.

Francis Ferraro, Max Thomas, Matthew R Gormley, Travis Wolfe, Craig Harman, and Benjamin Van Durme. 2014. Concretely annotated corpora. In *Proceedings of the AKBC Workshop at NIPS 2014*, Montreal, Canada, December.

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.

Maarten van Gompel and Martin Reynaert. 2013. FoLiA. A practical XML format for linguistic annotation. A descriptive and comparative study. *Computational Linguistics in the Netherlands Journal*, 3:63 – 81.

Thilo Götz and Oliver Suhre. 2004. Design and implementation of the UIMA Common Analysis System. *IBM Systems Journal*, 43(3):476 – 489.

Ulrich Heid, Helmut Schmid, Kerstin Eckart, and Erhard W. Hinrichs. 2010. A corpus representation format for linguistic web services. The D-SPIN Text Corpus Format and its relationship with ISO standards. In *Proceedings of the 7th International Conference on Language Resources and Evaluation*, page 494 – 499, Valletta, Malta.

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.

Nancy Ide, James Pustejovsky, Christopher Cieri, Eric Nyberg, Denise DiPersio, Chunqi Shi, Keith Suderman, Marc Verhagen, Di Wang, and Jonathan Wright. 2014a. The Language Application Grid. In *Proceedings of the 9th International Conference on Language Resources and Evaluation*, Reykjavik, Iceland.

Nancy Ide, James Pustejovsky, Keith Suderman, and Marc Verhagen. 2014b. The Language Application Grid Web Service Exchange Vocabulary. In *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT*, Dublin, Ireland.

ISO. 2012. Language Resource Management. Linguistic Annotation Framework. ISO 24612.

Emanuele Lapponi, Erik Velldal, Stephan Oepen, and Rune Lain Knudsen. 2014. Off-road LAF: Encoding and processing annotations in NLP workflows. In *Proceedings of the 9th International Conference on Language Resources and Evaluation*, page 4578 – 4583, Reykjavik, Iceland.

OMG. 2002. OMG XML metadata interchange (XMI) specification. Technical report, Object Management Group, Inc., January.

Marc Verhagen, Keith Suderman, Di Wang, Nancy Ide, Chunqi Shi, Jonathan Wright, and James Pustejovsky. 2016. The LAPPS Interchange Format. In *Revised Selected Papers of the Second International Workshop on Worldwide Language Service Infrastructure - Volume 9442*, WLSI 2015, pages 33–47, New York, NY, USA. Springer-Verlag New York, Inc.