A Lightweight Modeling Middleware for Corpus Processing

Markus Gärtner Jonas Kuhn
Institute for Natural Language Processing
University of Stuttgart
{markus.gaertner,jonas.kuhn}@uni-stuttgart.de

Abstract

Present-day empirical research in computational or theoretical linguistics has at its disposal an enormous wealth in the form of richly annotated and diverse corpus resources. Especially the points of contact between modalities are areas of exciting new research. However, progress in those areas in particular suffers from poor coverage in terms of visualization or query systems. Many limitations for such tools stem from the non-uniform representations of very diverse resources and the lack of standards that address this problem from the perspective of processing or querying. In this paper we present our framework for modeling arbitrary multi-modal corpus resources in a unified form for processing tools. It serves as a middleware system and combines the expressiveness of general graph-based models with a rich metadata schema to preserve linguistic specificity. By separating data structures and their linguistic interpretations, it assists tools on top of it so that they can in turn allow their users to more efficiently exploit corpus resources.

Keywords: data model, multi-modal, metadata

1. Introduction

The availability of richly annotated multi-layer corpora is one of the cornerstones of modern theoretical and computational linguistics. With a steady flow of newly created corpus resources and tools to annotate or process those corpora comes also the need for exploratory support tools, especially visualization and query systems, to get a better understanding of the resources at hand.

The context of this contribution is the successor project of the ICARUS (Gärtner et al., 2013) platform, an interactive visualization and query tool for corpora with dependency syntax, coreference structures (Gärtner et al., 2014) or rich annotations for intonation (Gärtner et al., 2015). This project aims at providing researchers a platform to explore and query even more diverse multi-layer and multi-modal corpus resources and artifacts through a single interactive interface. Our target resources include, but are not limited to, text and speech data annotated for a variety of different layers on varying linguistic levels and granularities, such as the SFB732 Silver Standard Collection, of which a first overview has been provided by Eckart and Gärtner (2016). A strong focus lies hereby on connecting linguistic layers or modalities which so far have not received much attention in their particular combination. This is to support interesting research questions, such as the interaction of information status and prosodic realization(Baumann and Riester, 2013) or the incorporation of prosodic information into the task of automatic coreference resolution on spoken text as done by Roesiger and Riester (2015).

As a result we faced the challenge of modeling access from a single processing software to a very diverse set of corpus resources. And while several standards around the *Linguistic Annotation Framework* (LAF)(Ide and Suderman, 2014) exist, they primarily take the point of view of preparation or curation of data. The ongoing standardization effort for a *Corpus Query Lingua Franca* (CQLF, ISO/DIS 24623-1)(Banski et al., 2016) correctly identifies the lack of standards which address a quite different view, that is, the one of querying or processing of corpus resources. As such there is no standardized or universally accepted solution readily available for this modeling task, which led us to designing and implementing a new dedicated framework.

With "modeling corpus resources" we refer to the task of representing the structure and content of a corpus or similar resource by means of a data model in memory, i.e. during an application's runtime. As a corpus or equivalent resource we treat any collection of utterances in arbitrary form, be they written, spoken or presented in yet another modality. Note that presently this definition excludes the modeling of other types of linguistic resources like lexicons or dictionaries as realized for example in the *Lexical Markup Framework* (Francopoulo et al., 2006).

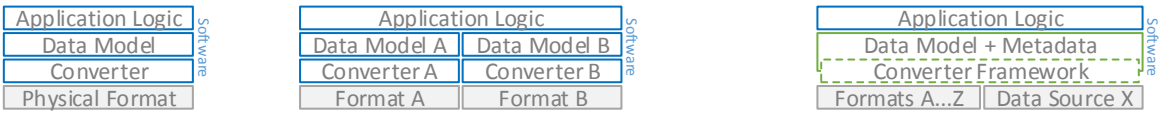
In the remainder of this paper we present our approach for providing unified access to very different corpus resources. Section 2 discusses the underlying issue of a heterogeneous format multiverse and Section 3 contextualizes our work. We present a detailed overview of various aspects of our framework in Section 4 and then proceed to illustrate its usage with code examples in Section 5. Information on the availability of the software and its documentation is provided in Section 6 and Section 7 concludes.

2. Format Multiverse and Middleware

Extensive work has already been invested into designing and establishing interoperable formats for linguistic resources and annotations based on various common transport formats. Notable examples are the *NLP Interchange Format* (NIF) (Hellmann et al., 2013), an RDF/OWL-based format, or *GrAF* (Ide and Suderman, 2007), a pivot XML-based serialization format for the *Linguistic annotation framework (LAF)* (Ide and Suderman, 2014). *GrAF* has also been standardized by ISO[1] as part of the "Language resource management" committee (ISO/TC 37/SC 4). The LAPPS Interchange Format (Verhagen et al., 2016) is a JSON-LD[2] format and used for data transfer between service implementations in the The Language Applications

[1] ISO 24612:2012

[2] https://json-ld.org/



(a) Monolithic design (b) Multiple supported format stacks (c) Middleware solution

Figure 1: Bottom-up data flows in several resource processing applications depicted as software stacks. Grey filled boxes represent data to be processed, blue bordered ones are parts of an application's own codebase and green shows the placement of our middleware solution in such an architecture in (c).

Grid (Ide et al., 2016). Less flexible exchange formats include for instance the *Text Corpus Format* (TCF) of the WebLicht (Hinrichs et al., 2010) environment, where the coverage of different annotation layers or hierarchical structures remains fixed.

Albeit having very expressive and standardized formats available, the reality is that more often than not corpora or system outputs used in day-to-day research work are stored in specialized or proprietary formats. Especially tools used for a specific task or field of research often use formats that have emerged as a kind of "local" standard for the particular field, such as Praat TextGrid files (Boersma, 2001) for phonetic analysis. Tools designed in the context of a particular task not uncommonly motivate the design and limitations of their data model with the original data format provided. In the popular CoNLL Shared Tasks for example data sets consistently are made available in a tabular format tailored to the individual tasks (e.g. parsing 2009 (Hajič et al., 2009), coreference resolution 2011 (Pradhan et al., 2011) and again parsing to Universal Dependencies in the 2017 shared task[3]). Tools developed for those tasks often made the respective CoNLL format their exclusive input and output representation, leading to a plethora of classic monolithic designs of the form shown in Figure 1a.

Processing applications such as search or visualization tools are then required to either support multiple formats and associated abstract models (see Figure 1b) or force the user into converting input data into a pivot format (further encouraging designs as in Figure 1a). This leads to increased workload for either the developers or end users. An alternative solution is to shift the unification effort into a framework that acts as a middleware between data sources and the actual processing logic of an application. The placement of our framework as this kind of middleware solution, in comparison to above situations, is presented in Figure 1c. Using a middleware solution also enables applications built on it to have their output freely converted into all formats supported by the framework. We believe that removing the need or motivation to build monolithic application architectures then in turn can even foster the use of standardized serialization formats like *GrAF*.

3. Related Work

At present several software toolkits for modeling corpus resources in a flexible way exist. Graph-based models have become a very popular approach for their implementation. *SALT* (Zipser, 2009), a graph-based model toolkit, is used in an established ecosystem together with the converter-framework *PEPPER* (Zipser et al., 2011) and the search interface *ANNIS3* (Krause and Zeldes, 2014), the latter combining it with a relational database for storage. *SALT* provides full bidirectional links between related nodes (such as a sentence and its tokens) and also imposes fixed restrictions on the scope of relations. It is well-suited for modeling shallow snapshots of a corpus, but especially the bidirectional linking makes it inherently expensive for modeling large numbers of concurrent structures (e.g. multiple parses for the same sentence) and hampers scalability. Motivated by the need to model rich annotations for text and speech data, the *NITE Object Model* (Carletta et al., 2003) is another graph-based model that allows linking of elements to portions of a singular shared timeline. Yet another approach is to use formalisms from the Semantic Web context like the Web Ontology Language (OWL[4]) and combine them with existing storage technology like XML or a relational database as proposed by Burchardt et al. (2008).

Based on the *Apache UIMA™project* a series of software components for natural language processing around the *DKPro Core* (Eckart de Castilho and Gurevych, 2014) framework have been developed. They are mainly focused on building shareable analysis pipelines and build on a strongly typed model, where linguistic types or theories are directly encoded. This can be helpful for data exchange between collections of predefined processing components such as parsers. However, it made this approach unsuited for us as alternative to designing our own modeling toolkit. Other text processing pipelines include *OpenNLP*[5], the *General Architecture for Text Engineering* (Cunningham et al., 2002) and the *Natural Language Toolkit* (Bird, 2006) for Python. They provide solutions for various NLP tasks and also feature modeling approaches of varying expressiveness. Their general focus on text data however keeps them limited and unfit as basis for a middleware system that attempts to unify access to multiple modalities.

4. Architecture

An implementation of our modeling approach is provided as an open-source Java toolkit (further information on availability is provided in Section 6) which requires version 8 of the Java Runtime Environment (JRE). It offers a set of abstract *building blocks* and other components to compose the data structures that make up a given corpus resource once read into the model and an extensive API to interact with them in various ways.

[3] http://universaldependencies.org/conll17

[4] www.w3.org/2004/OWL/

[5] http://opennlp.apache.org

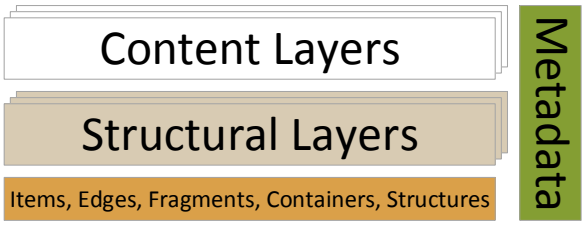


Figure 2: Core components for storage and associated metadata of the model architecture. Storage components are aligned horizontally.

A core concept is to separate components that represent addressable units (tokens, sentences, audio frames, syntactic structures, edges, regions of an image or video stream.) from their actual content (such as classic text annotations or binary data of audio or video streams). Figure 2 shows the hierarchy of core components in the model with parts responsible for storage aligned horizontally. The following sections line out the conceptual foundations of our approach and then describe the core components and concepts in detail.

4.1. Separation of Model Responsibilities

While the approaches described in Section 3 and many others are sufficient for the task of representing data in memory, the gain in representational generality is often achieved at the cost of specificity when it comes to the linguistic meaning or interpretation of modeled units or annotations. Thus even when using them as a developer one still needs to model this information and connect it with the data structures in such models.

To work around these shortcomings we decided to split the task of modeling corpus resources into two subparts:

One part for segmentation, structure and data storage maximizes generality but sacrifices any direct knowledge about linguistic specificity (equivalent to aforementioned general-purpose graph models). Independent of this the other part acts as metadata and carries the information describing composition and dependencies of a specific resource.

We outline the two aforementioned parts and how they interact with each other in Sections 4.3 to 4.5.

4.2. Design Foundations

Our model design builds on several observations and requirements for modeling diverse corpus resources, some of which have been proposed by Ide et al. (2003) in an early iteration of LAF. We list these requirements and show how our modeling approach fulfills them:

Expressive adequacy and Openness: We do not impose any limitations on the nature of linguistic information or theories that are expressed or stored in our framework to keep it as flexible as possible.

Uniformity: We utilize a compact set of universal "building blocks" (see Section 4.3) to model all kinds of structural or hierarchical information in a corpus (equal in expressiveness to a graph, but closer to linguistic universals).

Media independence: As stated in the introductory part of Section 4 our framework conceptually splits structural properties of a resource from its (media) content, treating all content as annotations. It thereby allows for arbitrary annotations to be associated with individual building blocks.

Semantic adequacy: We utilize a compact metadata schema (see Section 4.5) to formally define a resource's overall composition. This includes declarations of available layers as well as their hierarchies and dependencies.

Granularity: For every resource there is at least one mandatory segmentation layer that defines atomic units as a common foundation. This guarantees spacial comparability of all other structural constructs on top of it. As an extension to the original observation made by Ide et al. (2003) we also acknowledge the need to retrospectively define an even finer granularity as originally declared for a resource. Our framework supports subdivision of existing units in a corpus by declaring anchors into rastered[6] annotation values for those blocks (see Section 4.3 for further explanation and Section 5.3 for a usage example).

Extensibility: We offer various mechanisms for extending the functionality of the corpus modeling framework:

- A template system for the metadata schema to for example allow sharable tagset definitions or format schemata for converters (see Figure 4 for a tagset template).
- Integration of new converter implementations via the Java Plugin Framework[7], Java's own Service Provider Interface (SPI) or the Open Services Gateway initiative[8].
- An abstract type system for building blocks that allows custom implementations to optimize based on the underlying corpus resource, for instance read-only resources or when accessing a corpus stored in a (relational) database versus converting from classic file data.
- Interfaces to implement the handling of new media or annotation types for tasks such as fragmentation (cf. Section 4.3) or spacial comparison of those fragments.

Scalability: Special care is taken to ensure the framework and its performance are able to scale along two different and very important axes:

- *Horizontally* with the size of resources, i.e. the number of addressable units they contain (which can reach up to billions of tokens for web corpora like those from the *WaCky* family (Baroni et al., 2009)).
- *Vertically* in terms of the number of (parallel) annotation layers of arbitrary type. Especially speech corpora can contain very rich phonetic annotations on various levels when annotated by automatic systems.

4.3. Components

Instead of squeezing an entire corpus into one big graph we provide a series of types and constructs that model aspects of a corpus in ways closer to actual linguistic concepts.

[6] We borrow the term to describe the transformation of raw annotation values into collections of addressable discrete units, such as sequences or grids. Common examples are individual characters in a text, pixels in an image or frames in an audio stream.

[7] `http://jpf.sourceforge.net/`

[8] OSGi™ `https://www.osgi.org/`

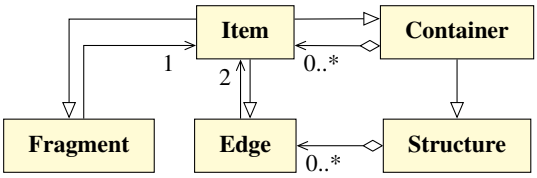


Figure 3: Simplified UML class diagram showing the relations between basic types described in Section 4.3.

Units: At the core we provide generic building blocks to model addressable units in a corpus (see Figure 3 for a basic UML class diagram of those types and their relations and Table 1 for additional explanations). Those `Items` can represent any individual data points, regions, edges or aggregations within a corpus. In addition to construct a corpus by aggregation, `Fragments` offer a way of subdividing existing items into smaller addressable units.
Each item in a corpus can be identified by two ways: One is a numerical *id* that is unique within its host layer and the other is its current *position* in that layer. While the id is persistent, the current index can change for editable corpus data and is mainly used for spatial comparisons or mappings. Unlike other approaches we do not maintain a full bidirectional linking between related items outside of their native host container. Instead a container only holds links to its contained items, but the reverse linking is moved into dedicated mapping facilities on the layer level. This keeps linking overhead to a minimum and makes the model scalable wrt concurrent segmental or structural information.

Layer: Layers in our model directly correspond to their linguistic counterparts and organize parts of a corpus such as annotations, segmentations, hierarchies or other types of structures. Layers carrying segmental or grouping information play a special role since they can act as foundation or boundary layers. The former provide a shared space for spatial comparisons equivalent to timelines in other systems and the latter are used for restricting the scope of complex constructs such as relations. To link different aggregating or foundation layers dedicated mapping facilities are used to translate between those index spaces.

Context: Layers are naturally grouped according to the data source their content originated from, for example a database, web-service or a local file. On the context level dependencies to external corpus resources are defined and each context must have a singular layer designated for the roles of *foundation* and *primary*, to define the segmentation of traversable data chunks of the context and their shared address space. Each context is associated with its individual converter implementation that mediates between the physical source of data and the in-memory model instances as described in more detail in Section 4.7.

Corpus: On the top of the hierarchy a corpus combines an arbitrary number of contexts, which enables modeling of resources that consist of multiple data sources or are expressed in several different formats. Each corpus in the framework is an independent and interactive object that client code can work with in many different ways, some of which are described and compared in Section 4.6.

Type	Function
Item	basic addressable unit
Edge	link/relation between items
Fragment	anchors used to split existing items based on their rastered annotation values
Container	logical grouping of items
Structure	container that augments its elements (= nodes) with edges

Table 1: Basic types used to model addressable units, segmentation and structure in a corpus resource and their respective functions. Aggregating types are italicized.

4.4. Annotations

A very central requirement for successfully unifying richly annotated corpus resources across different modalities is the ability to deal with very diverse types of annotations. To achieve this we impose no direct limitations on the nature of annotation values that can be stored or retrieved through the model. In addition we defined a (extensible) collection of commonly used annotation types that are directly supported by specialized storage implementations. The most important ones are listed as follows:

- Character sequences (`Strings`). With Java natively modeling characters as UTF-16 code units in memory, this directly provides sufficient Unicode support.
- Numerical values (`integer`, `long`, `float`, `double`)
- Links to external resources such as files or websites (`URL`, `URI` and local file paths).
- Images modeled as bitmap graphics.
- Binary streams such as audio or video content. Our framework does not restrict or support specific multimedia formats. It is left to the producer of a resource description (cf. Section 4.5) to include sufficient information about the encoding, preferably in the form of standardized format identifiers so that Java's default frameworks for handling media content can be used.
- References to other `Items` to model simple structures such as arguments.
- Vector and matrix types to allow aggregation of other supported value types.

4.5. Metadata

In Section 4.3 we covered the design of a very flexible data model for representing the content of arbitrary corpus resources. This maximized generality reduces the typical overhead of developing very content specific software components (e.g. a custom-built search or visualization interface tailored to the content of a single corpus).
Sacrificing specificity, if not compensated somehow, can on the other hand create certain disadvantages, such as not knowing how to visualize a certain piece of data expressed in the model. This is especially true for interactive systems that wish to provide assistive functions or information to the end-user. Such assistive functions are for example optimized visualization of linguistic data or automatically limiting user input for search constraints based on the con-

```
1  <annotation id="common.tags.stts" name="STTS−
   Tagset", valueType="string">
2   <values>
3    <value name="Adjective">JJ</value>
4    <value name="Noun">NN</value>
5    <value name="Determiner">DT</value>
6    <value name="Verb, gerund">VBG</value>
7    <value name="Verb, 3rd sg">VBZ</value>
8    [...]
9   </values>
10 </annotation>
```

Figure 4: Part of a manifest in XML format describing a part of the STTS tagset for part-of-speech annotations as a reusable template. See Figure 5 for an example of another metadata component referencing this template.

```
1  <corpus id="my.simple.corpus">
2   <context foundation="token">
3    <layerGroup primary="token">
4     <itemLayer id="token"/>
5     <itemLayer id="sentence">
6      <baseLayer layerId="token"/>
7      <container containerType="span"/>
8     </itemLayer>
9     <annotationLayer id="content">
10     <baseLayer layerId="token"/>
11     <annotation key="id" valueType="int"/>
12     <annotation key="form"/>
13     <annotation key="lemma"/>
14     <annotation key="pos" templateId="common.tags.
   stts">
15      <!-- Inherited from template -->
16     </annotation>
17    </annotationLayer>
18   </layerGroup>
19  </context>
20 </corpus>
```

Figure 5: Manifest XML specifying the structure of a simple two-level corpus resource with some shallow annotations. Note how the declaration for part-of-speech annotations in line 14 references a template previously defined in Figure 4. A sentence matching this structural description can be found in Figure 6 encoded in two different formats.

tent of a tagset. If only provided a general graph-like representation of a corpus without any additional information, a processing software component would have to undergo the time-consuming task of analyzing the resource first.

To this end we designed a compact metadata schema and its associated serialization format in XML that can express information regarding a corpus resource's structure and content. For every complex type (`Container` or `Structure`) and each layer used in the storage model this metadata model provides dedicated descriptions called **manifests** ("simple" types are described as part of their surrounding host construct). These descriptions include dependencies between layers, what annotations are available for which items, as well as types of structures or containers and more. For improved interoperability most metadata elements can be linked to established identifiers of linguistic categories to better express their meaning, for example using ISOcat/DatCatInfo as described in (Windhouwer and Wright, 2012). Conventional metadata to describe a resource's provenance is however outside the scope of this scheme.

Using such a manifest enables tools to properly handle the general data structures provided by the storage model totally independent from the original format or data source of a corpus. Figure 6 shows instances of two different textual formats that both encode the same chunk of data matching the description in Figure 5. For an application the resulting representation in our data model looks exactly the same. It could also contain information on how to access the resource it is describing, but this is currently outside our intended scope for the metadata model. Note that it remains the responsibility of the producer of a resource to provide this metadata and also to some extent to implement means of converting the raw form of a resource into an actual model representation (more on this in Section 4.7).

4.6. Interaction

Modeling frameworks often confine themselves to providing generic data structures and support for their serialization. This keeps them usually easy to use, but leaves the bulk of additional management to client code. Different types of applications pose very diverse requirements to underlying model libraries. To accommodate a wide range of possible use-cases our framework features different operation modes for interacting with a corpus:

Navigation: Depending on their individual needs, applications can access data in a corpus ***directly*** (communicating with the converter level without intermediate abstraction layers), as a ***stream*** (forward-only iteration) or in a "window" mode through so called ***views***. The latter approach allows for horizontal and vertical filtering (for instance to limit elements to candidates provided by some index system and to reduce the amount of required annotation layers) and also supports paging to traverse the selected sub-corpus block-wise.

Assistance: The components described in Section 4.3 can be used with a varying degree of framework support, for instance ***raw*** as generic building blocks if no management support by the framework is desired or ***fully managed*** when accessing the corpus content through a *view* or *stream*. Assistance includes for example structural or annotation verification based on the associated metadata definitions.

Editability: Resources are declared static or editable in their metadata and client code can additionally specify if it needs write access when connecting to a corpus. The framework is then able to optimize actual model instances for performance based on those decisions. When in *write* mode, the framework also offers an integrated edit history for live data to record (or undo) any modifications and a notification system to inform client code of changes.

4.7. Conversion

Of course the transformation of raw data into an in-memory model still requires a piece of software dedicated to that

```
1  Finding   find    VBG
2  the       the     DT
3  right     right   JJ
4  format    format  NN
5  is        be      VBZ
6  tricky    tricky  JJ
7  .         .       .
```

```
<s id="s1">
 <graph>
  <terminals>
   <t id="s1_1" word="Finding" lemma="find" pos="VBG"/>
   <t id="s1_2" word="the" lemma="the" pos="DT"/>
   <t id="s1_3" word="right" lemma="right" pos="JJ"/>
   <t id="s1_4" word="format" lemma="format" pos="NN"/>
   <t id="s1_5" word="is" lemma="be" pos="VBZ"/>
   <t id="s1_6" word="tricky" lemma="tricky" pos="JJ"/>
   <t id="s1_7" word="." lemma="." pos="."/>
  </terminals>
 </graph>
</s>
```

Figure 6: Automatically processed sentence in both a tabular (left text) format similar to the ones used in various CoNLL Shared Tasks, e.g. (Hajič et al., 2009), and TigerXML (König et al., 2003) on the right. Both representations contain the same annotations and hierarchical structure expressed by the manifest description in Figure 5.

particular representation or data source. This is an issue which cannot be solved universally due to the fundamental differences in the ways data is stored in different systems or file types (such as database system versus plain text or audio files). While ideally the existence of serialization formats for linguistic annotations such as *GrAF* (cf. Section 2) would obsolete many less expressive or flexible approaches, in reality pragmatic considerations of sticking to alternative formats often prevail. The matter is further complicated when taking different use-cases into account where corpora of vastly different sizes are involved and where aspects like pure storage-efficiency takes priority over those of expressiveness or flexibility of the format being used. As such the necessity of decoupling in-memory modeling completely from the original representations still remains.

In our framework this task of *physically* connecting to a given corpus resource (i.e. converting between its native form and the respective in-model instances) is performed by implementations of the `Driver` interface. Drivers allow the core framework to completely abstract away from the specifics of individual data sources. From the framework's point of view the complexity is reduced to querying the driver implementations for information such as the size of a layer and to issue read or write operations for selected parts of a resource (which in turn causes the driver to perform the necessary conversions).

While the initially stated complexity to some extent still remains, it is possible to implement this "converter" part of the application in such a way that the specifics of a physical representation can be defined by means of a schema which allows converters to adapt to input that is different but of a similar *type*[9] and greatly reduces required development effort. We cover a series of common formats and sources natively and provide generic converter solutions which can be configured using such schemata (either as part of a manifest, via external files or programmatically).

[9]For example the support for schema definitions to read arbitrary tabular text data in the ICARUS (Gärtner et al., 2015) exploration and query tool

5. Usage Examples

This section illustrates how client code uses the concepts introduced in previous sections and especially the different interaction modes outlined in Section 4.6. Where applicable, example code will assume the simple corpus structure shown in Figure 5 and the one-sentence content in Figure 6.

5.1. Building a Corpus

For low-level tasks, for instance a driver implementation, direct constructive interaction with the basic framework members is required. The following code snippet illustrates the construction of the sentence shown in Figure 6:

```
// Binding to manifests/storage
ContainerManifest sentenceManifest = ...;
Container sentenceRootContainer = ...;
AnnotationLayer contentLayer = ...;
// Simplify creation of building blocks via factory
LayerMemberFactory factory = newMemberFactory();
// Instantiate sentence container
Container sentence = factory.newContainer(
  sentenceManifest, sentenceRootContainer, 1);
// Create, add and annotate tokens
Item token1 = factory.newItem(sentence,1);
sentence.addItem(token1);
contentLayer.setValue(token1, "form", "Finding");

Item token2 = factory.newItem(sentence,2);
sentence.addItem(token2);
contentLayer.setValue(token2, "form", "the");

// Finally add finished sentence
sentenceRootContainer.addItem(sentence);
```

The process involves manually constructing the individual tokens, grouping them into a sentence and attaching annotations. It also assumes an existing host corpus and access to certain parts of the associated manifest and layers to store new content in. For brevity only the first two tokens are constructed and other annotation layers besides the `form` layer are omitted.

5.2. Accessing Corpus Parts

As mentioned previously in Section 4.6 there are several ways for accessing (parts of) a corpus resource through the framework. The following two code examples show very different approaches, one involving horizontal filtering and one for simple sequential streaming.
In the first snippet client code is only interested in a selected subset of tokens. It qualifies the actual token indexes by

means of `IndexSet` instances and the layers it requires by a `Scope` in lines 3 and 5. With those parameters the corpus then creates a `CorpusView` object which essentially acts as a filtered sub-corpus. Iterating over the tokens contained in this view and outputting their respective `form` annotations then results in "the format is tricky" to be printed one token per line:

```
1 Corpus corpus = ...;
2 // Define the section of interest
3 IndexSet[] indices = IndexUtils.wrap(1,3,4,5);
4 // Define the layers of interest
5 Scope scope = Scope.withLayers(corpus, "token",
  "content");
6 // Filtered "window" of the corpus
7 CorpusView view = corpus.createView(scope, indices,
  AccessMode.READ, Options.NONE);
8 // Our "triple store" of annotations
9 AnnotationLayer content = view.fetchLayer("content");
10 // Request loading of the corpus portion
11 view.getPageControl().load();
12 // Interface for accessing the view
13 CorpusModel model = view.getModel();
14 // Contained "list" of tokens
15 Container rootContainer = model.getRootContainer();
16 // Iterate over all tokens
17 for(int i=0; i<rootContainer.getItemCount(); i++) {
18   Item item = rootContainer.getItemAt(i);
19   System.out.println(content.getValue(item, "form"));
20 }
```

Often an application does not require elaborate filtering of a corpus before processing. Especially analysis tools such as parsers simply need a way of accessing an entire corpus one element at a time. For this use-case the framework provides streaming, as shown in the following code:

```
1 Corpus corpus = ...;
2 // Define the layers of interest
3 Scope scope = Scope.withLayers(corpus, "token",
  "content");
4 // Create stream for entire corpus
5 ItemStream stream = corpus.getStream(scope,
  AccessMode.READ, Options.NONE);
6 // Our "triple store" of annotations
7 AnnotationLayer content = view.fetchLayer("content");
8 // Traverse items in sequence
9 while(stream.advance()) {
10   Item item = stream.currentItem()
11   System.out.println(content.getValue(item, "form"));
12 }
```

Again client code defines the layers it is interested in, but this time it obtains an `ItemStream` that lets it iterate over all the tokens in the corpus. Since there is no horizontal filtering involved this time, the output will be the entire sentence from Figure 6 "Finding the right format is tricky .".

5.3. Fragmentation

What is the obligatory form for a corpus' primary data? Should it be a collection of already segmented units? Or should it be the original raw data such as the entire text of a book? Rarely is there an absolute answer for such questions within the universe of highly diverse corpus resources.

Our framework therefore does not force a singular approach for composing a corpus. The previous code snippets always used elements of `token` layer that already were properly segmented. The next example shows how those tokens can be created by splitting annotation values to fragment the respective original text.

```
1 // Simplify creation of building blocks
2 LayerMemberFactory factory = newMemberFactory();
3 // Existing item that represents entire text
4 Item text = ...;
5 // Define span for "Finding"
6 Position begin = Positions.create(0);
7 Position end = Positions.create(6);
8 Fragment token1 = factory.newFragment(sentence, 1,
  text, begin, end);
9 // Define span for "the"
10 begin = Positions.create(8);
11 end = Positions.create(10);
12 Fragment token2 = factory.newFragment(sentence, 2,
  text, begin, end);
```

5.4. Using Manifests

Manifests (the metadata part of our framework) offer a great way for applications to optimize computation or graphical user interfaces specifically for the content of a corpus in advance. Since the entire composition of a corpus object is described in detail in a formalized manner by its associated manifest, client code can use this information without having to actually read the content of a the corpus or to tailor its behavior to a certain type of resource.
The following code snipped illustrates how an application can use the manifest in Figure 5 (more specifically, the tagset template from Figure 4) to create specialized components for its user interface (UI):

```
1 // Manifest as provided to client code
2 AnnotationLayerManifest layerManifest = ...;
3 // Pick part-of-speech annotations as example
4 AnnotationManifest annotations =
  layerManifest.getAnnotationManifest("pos");
5 // Fetch the allowed values for this annotation
6 ValueSet tagset = annotations.getValueSet()
7 for(int i=0; i<tagset.valueCount(); i++) {
8   Object value = tagset.getValueAt(i);
9   // Check if advanced documention is available
10  if(value instanceof ValueManifest) {
11    ValueManifest manifest = (ValueManifest) value;
12    // Human readable id of the tag
13    String name = manifest.getName();
14    // Detailed description of the tag
15    String description = manifest.getDescription();
16    // Actual underlying tag value
17    value = manifest.getValue();
18    // Now make a more user friendly UI
19    addComplexUiElement(value, name, description);
20  } else {
21    // Use "value" without additional info
22    addSimpleUiElement(value);
23  }
24 }
```

By exploiting the information available through the predefined tagset, the application can create rich UI components that provide increased usability to its users. For instance, human readable names and descriptions that accompany the bare value definitions in the manifest can be used for additional info-labels or tooltips in the interface.

6. Availability

The core of our framework is available as a collection of individual Java libraries that cover and implement different aspects of our approach. We published the model libraries themselves, as well as a detailed documentation in the framework of CLARIN[10] and made it available via a persistent identifier[11] in order to ensure sustainability.

[10] `https://www.clarin.eu/`

[11] `http://hdl.handle.net/11022/1007-0000-0007-C636-D`

7. Conclusion

In this paper we presented our design and implementation of a novel corpus modeling framework. It approaches the task of modeling corpus resources from a perspective of processing and querying. Being available as a Java toolkit it can be used to provide unified in-memory representations of arbitrary corpus resources. Independence wrt to linguistic theories or tagsets allows it to work across different modalities. While our reference implementation is provided in Java, the default metadata serialization format is XML and so the entire concept can be also transfered to other programing languages.

In the future we plan on broadening the support for (standardized) serialization formats and also provide a built-in interface for database connectivity. While currently focused strictly on the modeling of corpora as collections of utterances, we do consider extending the toolkit to at least conceptually recognize other resources such as lexicons for which adequate modeling solutions already exist.

8. Acknowledgements

This work was funded by the German Federal Ministry of Education and Research (BMBF) via CLARIN-D, No. 01UG1120F and the German Research Foundation (DFG) via the SFB 732, project INF.

9. Bibliographical References

Banski, P., Frick, E., and Witt, A. (2016). Corpus query lingua franca (cqlf). In Nicoletta Calzolari (Conference Chair), et al., editors, *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, Paris, France, may. European Language Resources Association (ELRA).

Baroni, M., Bernardini, S., Ferraresi, A., and Zanchetta, E. (2009). The WaCky wide web: a collection of very large linguistically processed web-crawled corpora. *Language Resources and Evaluation*, 43(3):209–226.

Baumann, S. and Riester, A. (2013). Coreference, lexical givenness and prosody in German. *Lingua*, 136:16–37.

Bird, S. (2006). NLTK: The Natural Language Toolkit. In *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*, pages 69–72, Sydney, Australia, July. Association for Computational Linguistics.

Boersma, P. (2001). Praat, a system for doing phonetics by computer. *Glot International*, 5(9/10):341–345.

Burchardt, A., Padó, S., Spohr, D., Frank, A., and Heid, U. (2008). Constructing integrated corpus and lexicon models for multi-layer annotations in OWL DL. *Linguistic Issues in Language Technology*, 1:1–33.

Carletta, J., Kilgour, J., O'Donnell, T., and O'donnell, T. (2003). The nite object model library for handling structured linguistic annotation on multimodal data sets. In *In Proceedings of the EACL Workshop on Language Technology and the Semantic Web (3rd Workshop on NLP and XML, NLPXML-2003*, page 2003.

Cunningham, H., Maynard, D., Bontcheva, K., and Tablan, V. (2002). Gate: An architecture for development of robust hlt applications. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 168–175, Stroudsburg, PA, USA. Association for Computational Linguistics.

Eckart, K. and Gärtner, M. (2016). Creating Silver Standard Annotations for a Corpus of Non-Standard Data. In Stefanie Dipper, et al., editors, *Proceedings of the 13th Conference on Natural Language Processing (KONVENS 2016)*, volume 16 of *BLA: Bochumer Linguistische Arbeitsberichte*, pages 90–96, Bochum, Germany.

Eckart de Castilho, R. and Gurevych, I. (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.

Francopoulo, G., George, M., Calzolari, N., Monachini, M., Bel, N., Pet, Y., and Soria, C. (2006). Lexical markup framework (lmf. In *In Proceedings of LREC2006*.

Gärtner, M., Thiele, G., Seeker, W., Björkelund, A., and Kuhn, J. (2013). ICARUS – an extensible graphical search tool for dependency treebanks. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Sofia, Bulgaria, August. Association for Computational Linguistics.

Gärtner, M., Björkelund, A., Thiele, G., Seeker, W., and Kuhn, J. (2014). Visualization, search, and error analysis for coreference annotations. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 7–12, Baltimore, Maryland, June. Association for Computational Linguistics.

Gärtner, M., Schweitzer, K., Eckart, K., and Kuhn, J. (2015). Multi-modal visualization and search for text and prosody annotations. In *Proceedings of ACL-IJCNLP 2015 System Demonstrations*, pages 25–30, Beijing, China, July. Association for Computational Linguistics and The Asian Federation of Natural Language Processing.

Hajič, J., Ciaramita, M., Johansson, R., Kawahara, D., Martí, M. A., Màrquez, L., Meyers, A., Nivre, J., Padó, S., Štěpánek, J., Straňák, P., Surdeanu, M., Xue, N., and Zhang, Y. (2009). The conll-2009 shared task: Syntactic and semantic dependencies in multiple languages. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task*, CoNLL '09, pages 1–18, Stroudsburg, PA, USA. Association for Computational Linguistics.

Hellmann, S., Lehmann, J., Auer, S., and Brümmer, M., (2013). *Integrating NLP Using Linked Data*, pages 98–113. Springer Berlin Heidelberg, Berlin, Heidelberg.

Hinrichs, E. W., Hinrichs, M., and Zastrow, T. (2010). WebLicht: Web-Based LRT Services for German. In *Proceedings of the ACL 2010 System Demonstrations*, pages 25–29.

Ide, N. and Suderman, K. (2007). GrAF: A graph-based format for linguistic annotations. In *Proceedings of the Linguistic Annotation Workshop*, pages 1–8, Prague,

Czech Republic, June. Association for Computational Linguistics.

Ide, N. and Suderman, K. (2014). The linguistic annotation framework: A standard for annotation interchange and merging. *Lang. Resour. Eval.*, 48(3):395–418, September.

Ide, N., Romary, L., and de la Clergerie, E. (2003). International standard for a linguistic annotation framework. In Jon Patrick et al., editors, *Proceedings of the HLT-NAACL 2003 Workshop on Software Engineering and Architecture of Language Technology Systems (SEALTS)*, pages 25–30.

Ide, N., Pustejovsky, J., Cieri, C., Nyberg, E., DiPersio, D., Shi, C., Suderman, K., Verhagen, M., Wang, D., and Wright, J., (2016). *The Language Application Grid*, pages 51–70. Springer International Publishing, Cham.

König, E., Lezius, W., and Voormann, H., (2003). *TIGERSearch 2.1 User's Manual. Chapter V - The TIGER-XML treebank encoding format*. IMS, Universität Stuttgart.

Krause, T. and Zeldes, A. (2014). Annis3: A new architecture for generic corpus query and visualization. *Digital Scholarship in the Humanities*.

Pradhan, S., Ramshaw, L., Marcus, M., Palmer, M., Weischedel, R., and Xue, N. (2011). CoNLL-2011 Shared Task: Modeling Unrestricted Coreference in OntoNotes. In *CoNLL: Shared Task*, pages 1–27, Portland, Oregon, USA, June.

Roesiger, I. and Riester, A. (2015). Using prosodic annotations to improve coreference resolution of spoken text. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 83–88, Beijing, China, July. Association for Computational Linguistics.

Verhagen, M., Suderman, K., Wang, D., Ide, N., Shi, C., Wright, J., and Pustejovsky, J., (2016). *The LAPPS Interchange Format*, pages 33–47. Springer International Publishing, Cham.

Windhouwer, M. and Wright, S. E. (2012). Linking to linguistic data categories in isocat. In *Linked Data in Linguistics*, pages 99–107. Springer.

Zipser, F., Zeldes, A., Ritz, J., Romary, L., and Leser, U. (2011). Pepper: Handling a multiverse of formats.

Zipser, F. (2009). Entwicklung eines Konverterframeworks für linguistisch annotierte Daten auf Basis eines gemeinsamen (Meta-)modells, November. In this diploma thesis I present a framework to convert linguistic data coming from a specific linguistic format to other linguistic formats. This approach is based on a common meta-model, which is used as an intermediate step to decrease the number of necessary mappings.