This document is part of EXCITEMENT project,
funded by the 7th Framework Programme of the European Commission through grant agreement no.: 287923.

Exploring Customer Interaction via Textual entailMENT

Deliverable 3.1a: Specifications and architecture for the open platform, I. cycle

Authors: Tae-Gil Noh, Sebastian Pado, Ofer Bronstein, Asher Stern, Rui Wang, Roberto Zanoli

Dissemination Level: Public

Date: June 30th, 2012

Grant agreement no.	287923
Project acronym	EXCITEMENT
Project full title	Exploring Customer Interaction via Textual entailMENT
Funding scheme	STREP
Coordinator	Moshe Wasserblat (NICE)
Start date, duration	1 January 2012, 36 months
Distribution	Public
Contractual date of delivery	30/06/2012
Actual date of delivery	30/06/2012
Deliverable number	3.1a
Deliverable title	Specifications and architecture for the open platform, I. cycle
Type	Report
Status and version	Final
Number of pages	80
Contributing partners	BIU, DFKI, FBK, HEI
WP leader	HEI
Task leader	HEI
Authors	Tae-Gil Noh, Sebastian Pado, Ofer Bronstein, Asher Stern, Rui Wang, Roberto Zanoli
EC project officer	Philippe Gelin
The partners in EXCITEMENT are:	NICE systems, Israel
	Fondazione Bruno Kessler (FBK), Italy
	Bar-Ilan University, Israel
	Heidelberg University, Germany
	Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Germany
	OMQ, Germany
	ALMA, Italy

For copies of reports, updates on project activities and other EXCITEMENT-related information, contact:

NICE Systems
EXCITEMENT
Moshe Wasserblat moshew@nice.com
Hapnina 8 Phone: +972 (9) 775-3702
Ra'anana, Israel Fax: +972 (9) 775-3702

Copies of reports and other material can also be accessed via http://www.excitement-project.eu

Table of Contents

Appendixes

1. Introduction

Identifying semantic inference relations between texts is a major underlying language processing task, needed in practically all text understanding applications. For example, Question Answering and Information Extraction systems should verify that extracted answers and relations are indeed inferred from the text passages. While such apparently similar inferences are broadly needed, there are currently no generic semantic *inference engines*, that is, platforms for broad textual inference.

Annotation tools do exist for narrow semantic tasks (i.e. they consider one phenomenon at a time and one single fragment of text at time). Inference systems assemble and augment them to obtain a complete inference process. By now, a variety of architectures and implemented systems exist, most at the scientific prototype stage. The problem is that there is no standardization across systems, which causes a number of problems. For example, reasoning components cannot be re-used, nor knowledge resources exchanged. This hampers in particular the pick-up of textual entailment as a "standard" technology in the same way that parsing is used, by researchers in semantics or NLP applications.

One of the primary scientific goals of the EXCITEMENT project is to address this situation by developing a generic inference engine platform. We envisage its role to be similar to the one played by MOSES in the Machine Translation community -- that is, as a basis for reusable development. Our goal is to realize the following properties:

1. *Algorithm independence.* The platform should be able to accommodate a wide range of possible algorithms (or more generally, strategies) for entailment inference.

2. *Language independence.*Similarly, the platform should be, as far as possible, agnostic to the language that is processes so that it can be applied to new languages in a straightforward manner.

3. *Component-based architecture.* We base our approach onto the decomposition of entailment inference decisions into a set of more or less independent components which encapsulate some part of the entailment computation -- such as the overall decision and different kinds of knowledge. Since components communicate only through well-defined interfaces, this architecture makes it possible to combine the best possible set of components given the constraints of a given application scenario, and the re-use of components that have been developed.

4. *Versatility.* The platform should be configurable in different ways to meet the needs of different deployment scenarios. For example, for deployment in industrial settings, efficiency will be a primary consideration, while research applications may call for a focus on precision.

5. *Clear specification and documentation.*

6. *Reference implementation.* We will provide an implementation of the platform that covers a majority of proposed entailment decision algorithms together with large knowledge resources for English, German, and Italian.

For reasons of practicality, the implementation will be based on three preexisting systems for textual entailment. These systems are:

- *BIUTEE*, The Bar Ilan University Textual Entailment Engine (BIU).
- *EDITS*, an edit distance-based approach to textual entailment recognition (FBK).
- *TIE*, Textual Inference Engine (DFKI).

The role of this document is to meet the third goal -- to provide a specification of the EXCITEMENT platform. Our goal for the specification is to be as general as possible (in order to preserve the goal of generality) while remaining as specific as necessary (to ensure compatibility). At a small number of decisions, we have sacrificed generality in order to keep the implementation effort manageable; these decisions are pointed out below.

The structure of this document mirrors the aspects of the EXCITEMENT platform that require specification. These aspects fall into two categories: actual interfaces, and specification of meta-issues.

Regarding interfaces, we have to specify:

a. The linguistic analysis pipeline which creates some data structure with linguistic information which serves as the input for the actual entailment inference (Section 3).

b. The interfaces between components and component groups within the actual entailment computation (Section 4).

As for meta-issues, we need to address:

c. The shape of the overall architecture and the definition of terminology (Section 2).

d. Further standardization issues such as configuration, shared storage of resources, error handling etc. (Section 5)

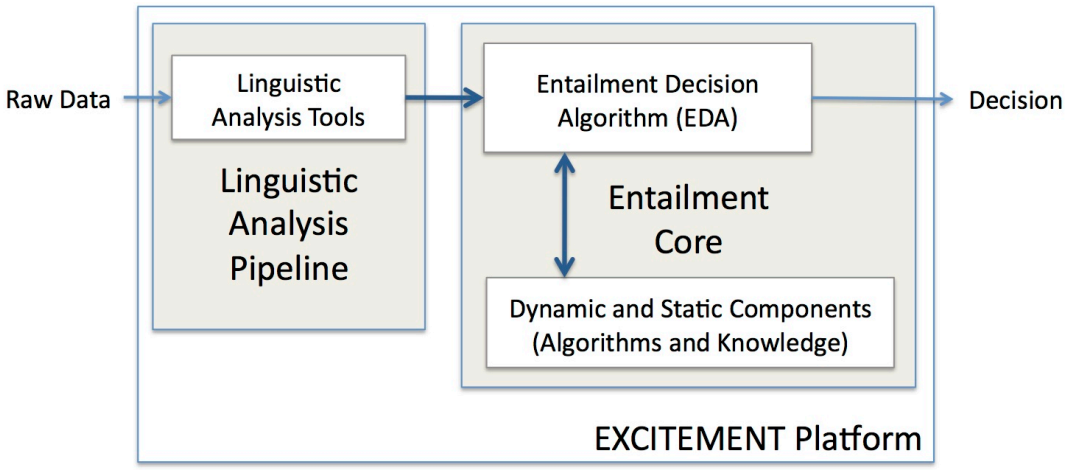
Currently outside the scope of this document is the specification of the transduction layer (to be added in a later version of this document).

2. Elements of the EXCITEMENT platform

2.1. The Platform

As sketched, we assume that it is beneficial to decompose the recognition of textual entailment into individual components which are best developed and independent of each other. This approach gives rise to the overall system architecture that is shown in Figure 1, “The EXCITEMENT platform”.

Figure 1. The EXCITEMENT platform



The most important top-level distinction is between the *Linguistic Analysis Pipeline (LAP)* and the *Entailment Core*. We separate these two parts in order to (a), on a conceptual level, ensure that the algorithms in the Entailment Core only rely on linguistic analyses in well-defined ways; and (b), on a practical level, make sure that the LAP and the Entailment Core can be run independently of one another (e.g., to preprocess all data beforehand).

Since it has been shown that deciding entailment on the basis of unprocessed text is a very difficult endeavour, the Linguistic Analysis Pipeline is essentially a series of annotation modules that provide linguistic annotation on various layers for the input. The Entailment Core then performs the actual entailment computation on the basis of the processed text.

The Entailment Core itself can be further decomposed into exactly one *Entailment Decision Algorithm (EDA)* and zero or more *Components*. An Entailment Decision Algorithm is a special Component which computes an entailment decision for a given Text/Hypothesis pair. Trivially, each complete Entailment Core is an EDA. However, the goal of EXCITEMENT is exactly to identify functionality within Entailment Cores that can be re-used and, for example, combined with other EDAs. Examples of functionality that are strong candidates for Components are WordNet (on the knowledge side) and distance computations between text and hypothesis (on the algorithmic side). Both of these can be combined with EDAs of different natures, and should therefore be encapsulated in Components.

2.1.1. Engines: Instantiations of the Platform

In order to use the EXCITEMENT infrastructure, a user will have to configure the platform for his application setting, that is, for his language and his analysis tools (on the pipeline side) and for his algorithm and components (on the entailment core side).

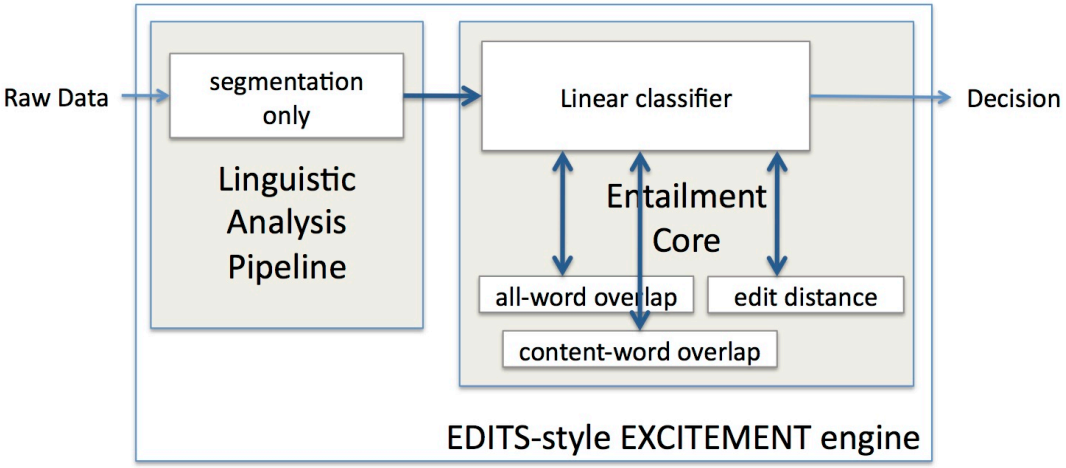
For an example, consider Figure Figure 2, “An EDITS-style EXCITEMENT engine”. It shows an engine instantiating the EXCITEMENT platform that mirrors the functionality of the EDITS system. The linguistic analysis pipeline remains very basic and only provides fundamental tasks (sentence segmentation and tokenization). The components are different variants of simple distance and overlap measures (at the

string token level) between text and hypothesis. The EDA is a simple linear classifier which computes weighted output of the components into a final score and compares it against a threshold.

This is a supervised learning setup -- both the weights for components and the threshold must be learned. Therefore, training must be part of the EXCITEMENT platform's functionality.

Note also that although this engine is fairly generic, it is not completely language-independent. Language-specific functionality is definitely part of the content word overlap component (at least in the form of a "closed class word list") and potentially also in the linguistic analysis pipeline, for example in the form of tokenization knowledge. For this reason, the LAP must enrich the input with meta data that describes its language, as well as the performed preprocessing steps so that the Entailment Core can verify the applicability of the current Engine configuration to the input data.

Figure 2. An EDITS-style EXCITEMENT engine



2.2. Concepts and Terminology

2.2.1. Key words

The key words *must*, *must not*, *required*, *shall*, *shall not*, *should*, *should not*, *recommended*, *may*, and *optional* are to be interpreted as described in [RFC 2119]. Note that for reasons of style, these words are not capitalized in this document.

2.2.2. Key Terms (Content)

Entailment Platform
: The totality of the infrastructure provided by the EXCITEMENT project for all languages and entailment paradigms. The Platform can be configured into an Entailment Engine.

Entailment Engine
: A configured version of the Entailment Platform that makes entailment decisions for one language. An Entailment Engine consists of a Linguistic Analysis Pipeline and an Entailment Core.

Linguistic Analysis Pipeline (LAP)
: A set of linguistic tools that analyze a given a set of Text-Hypothesis pairs, typically performing steps such as sentence splitting, POS tagging, Named Entity Recognition etc.

Entailment Core
: A part of an Entailment Platform that decides entailment for a set of Text-Hypothesis pairs that have been processed in the Linguistic Analysis Pipeline. The Entailment Core consists of exactly one Entailment Decision Algorithm and zero or more Components.

Entailment Decision Algorithm (EDA)
: An Entailment Decision Algorithm is a Component which takes a Text-Hypothesis pair and returns one of a small set of answers. A complete entailment recognition system is trivially an EDA. However, in the interest of reusability, generic parts of the system should be made into individual Components. Entailment Decision Algorithms communicate with Components through generic specified interfaces.

Component
: Any functionality that is part of an entailment decision and which is presumably reusable. This covers both "static" functionality (that is, knowledge) and "dynamic" functionality (that is, algorithms). A typology of Components is given in Section 4.1.3, “Identifying common components and interfaces ”.

User code
: Any code that calls the interfaces and utilizes the types defined in this specification. This includes the "top level" entailment code that calls the LAP and the EDA to apply an engine to an actual dataset.

2.2.3. Key Terms (Methodology)

Interface
: A set of methods and their signatures for a class that define how the class is supposed to interact with the outside world

Type
: We use the term "type" for classes that denote data structures (i.e. which have a representational, rather than algorithmic, nature).

Contract
: Further specification on how to use particular interfaces and types that goes beyond signatures. For example, rules on initialization of objects or pre/postconditions for method calls.

3. Linguistic Analysis Pipeline (LAP)

This section describes the specification (interfaces and exchange data structure) of the Linguistic Analysis Pipeline.

3.1. Requirements

This subsection describes the requirements for two aspects of the LAP: First, the user interface of the LAP. Second, the type (i.e., data structure) that is used to exchange data between the LAP and the Entailment Core.

3.1.1. Requirements for the linguistic analysis pipeline

- *Separation between LAP and entailment core.* (This is a global requirement.)
- *Language independence.* The pipeline should not be tied to properties of individual languages.
- *"One-click analysis".* The pipeline should be easy to use, ideally runnable with one command or click.
- *Customizability.* The pipeline should be easily extensible with new modules.
- *Easy configuration.* The pipeline should be easy to (re)configure.

3.1.2. Requirements for the interface between LAP and entailment core

- *Language independence.* The data structure should be able to accommodate analyses of individual languages.
- *Theory independence.* The data structure should be independent of specific analysis paradigms.
- *Extensible multi-level representation.* The data structure should be extensible with new linguistic information when the LAP is extended with new modules.
- *Support for in-memory and on-disk storage.* The data structure should be serializable so that LAP and Entailment Core can be run independently of one another.
- *Metadata support.* The data structure should encode metadata that specify (a) the language of the input data and (b) the levels and type of linguistic analysis so that the Entailment Core can ensure that the input meets the requirements of the current Engine configuration.

3.1.3. A rejected option: CoNLL

An option which we initially considered was the CoNLL shared task tabular format (http://ufal.mff.cuni.cz/conll2009-st/task-description.html). We rejected this possibility because it could not meet several of our requirements. (1), it does not specify a pipeline, just an interchange format. (2), it is extensible but at the same time fairly brittle and unwieldy for more complex annotations (cf. the handling of semantic roles in CoNLL 2009 which requires a flexible number of columns). (3), it only specifies an on-disk file format but no in-memory data structure. (4), no support for metadata.

3.2. UIMA as linguistic analysis pipeline of EXCITEMENT

3.2.1. UIMA

UIMA (Unstructured Information Management Applications) is a framework that started as a common platform for IBM NLP components. It evolved into a well-developed unstructured information processing

platform, which is now supported by Apache Foundation as an open source framework. It has been used by many well known NLP projects and has healthy communities in both academic and commercial domain.

UIMA provides a unified way of accessing linguistic analysis components and their analysis results. All analysis modules are standardized into UIMA components. UIMA components shares a predefined way of accessing input, output, configuration data, etc. Within UIMA it is easy to setting up a composition of analysis components to provide new pipelines, or adopt a newly developed analysis modules to already existing analysis pipeline.

On the top level, the unification of UIMA components is achieved on two levels. The first is unification of components behavior. Instead of providing different APIs for each analysis module, UIMA components all shares a set of common methods. Also, calling and accessing of a component is done by the UIMA framework, not by user level codes. Users of a component do not directly call the component. Instead, they request the UIMA framework to run the component and return the analysis result. The framework then calls the component with predefined access methods contracted among UIMA components. This common behavior makes it possible to treat every component as pluggable modules, and enables the users to establish two or more components to work together.

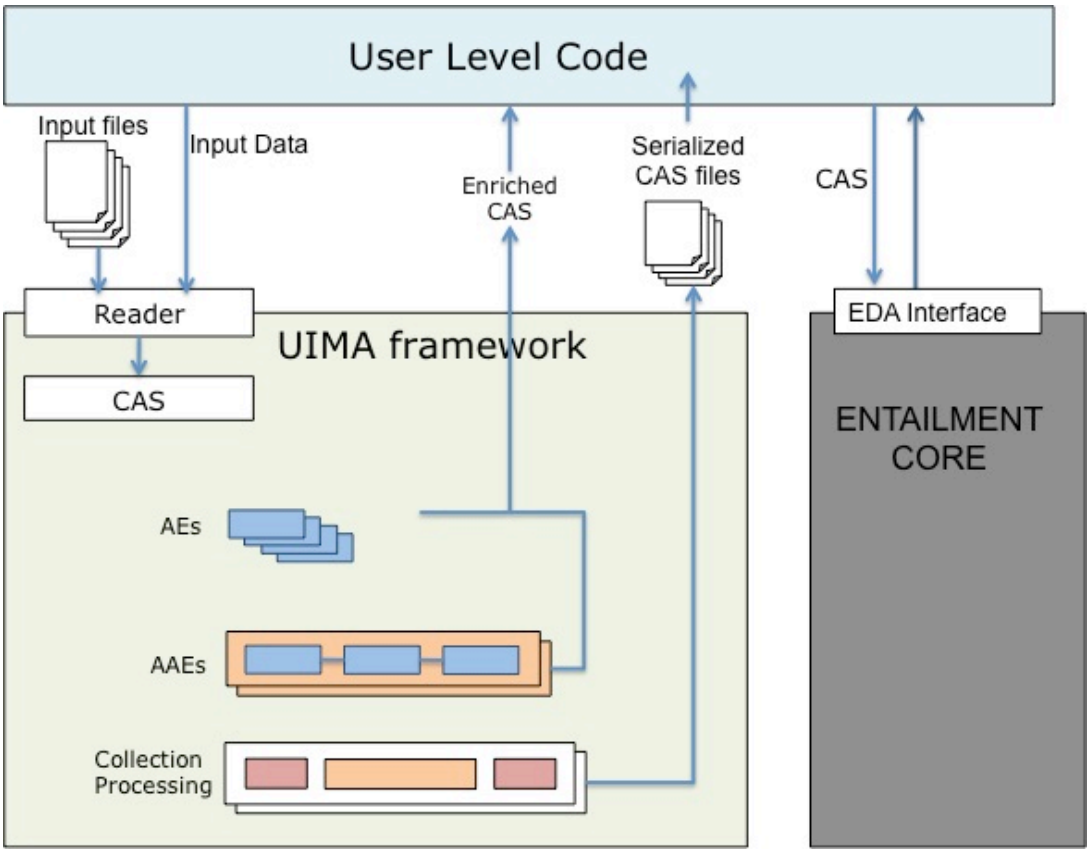
The second unification is done with common data representation. If the inputs and outputs of each component is not compatible, then the unification of component's behaviors are not really meaningful. In UIMA, compatibility of inputs and outputs are provided by the Common Analysis Structure (CAS) [UIMA-CAS]. One can see CAS as a data container, which holds original texts with various layers of analysis results that annotates the original text. As a CAS is passed through analysis engines, additional annotation layers are stacked into the CAS. Input and output of every components are standardized in terms of the CAS. Every component works on the CAS as the input, and adds something to the CAS as output. Thus, within UIMA, each components capabilities can be described in terms of CAS; what it needs in the CAS to operate (for example, a parser might need POS analysis result), and what it will add on the CAS after the processing (like dependency parse tree).

CAS itself is a container. The actual content (analysis result) is defined by the CAS type systems. UIMA provides a generic type system that is strong enough to describe various annotation layers (like POS tagging, parsing, coreference resolution, etc). Definition and usage of common type system is one important aspect of using CAS. The UIMA framework itself only provides a handful of default types like strings, and double values. Additional types must be proposed or adopted for actual systems; we can however cover most of our requirements with existing type systems.

3.2.2. Capabilities provided by the EXCITEMENT linguistic analysis pipeline

The following figure shows the Apache UIMA Java implementation as the LAP of EXCITEMENT. It provides a unified way of calling linguistic analysis components for user code.

Figure 3. UIMA as the linguistic analysis pipeline of EXCITEMENT



UIMA comes with various components and tools . However, in this specification, we will only review the components that an EDA developer should provide to the EXCITEMENT LAP as UIMA components. They are also the components that EXCITEMENT top level users can access from the linguistic analysis pipeline. Figure 3, "UIMA as the linguistic analysis pipeline of EXCITEMENT " shows them. We first provide generic definition of AE, AAE and collection processing. Then we will state what EDA implementers should provide to realize an EXCITEMENT LAP.

AE. Analysis modules of UIMA are called as Analysis Engines (AEs). For example, individual analysis components like POS tagger, parser, NER annotators are AEs. All AEs shares a set of contracted methods, and they will be accessed by the UIMA framework. All analysis results of a AE are returned to the caller as added layers on a CAS.

AAE. UIMA also provides analysis pipelines that are called Aggregated Analysis Engines (AAEs). An AAE is defined by composing multiple AEs. For example, a pipeline that outputs parsing results can be composed by subsequent calling of sentence splitter AE, POS tagger AE, and parser AE. UIMA provides standard way of composing analysis pipelines without changing anything on the individual analyzers.

Collection Processing (and UIMA Runtime). Linguistic analysis pipeline often needs to work on a collection of documents and produce a set of enriched documents. For example, a user might want to optimize parameters of a TE system with repeated experiments. In this case, repeating the same linguistic preprocessing is unneeded redundancy. AAEs do not provide any capability on accessing files, or processing collections. Thus, such file reading or calling multiple times on each file, are AAE callers role to fill in. However, in such a case, we cannot expect the same behavior among collection processors. To provide common behavior of collection processing, UIMA provides *UIMA runtime* and *collection reader*.

A *collection reader* is a UIMA component that is specialized in data reading and CAS generation. Compared to an AE, it shares a different set of contractual methods. A collection reader reads an input (document) and generates a CAS that can be processed by AEs.

A *UIMA runtime* is a runner mechanism that enables running of AEs in a separate space (other than callers space). If a user calls an AE directly (via framework), the component is actually running on the callers space (same process, thread). In this case, running AEs on multiple files would still be the callers responsibility. A UIMA runtime, is designed to take over that responsibility. A runtime is capable of iterating over a set of documents in its own space. Apache UIMA and UIMA community provides various runtimes, from simple single thread runtime to asynchronous distributed runtimes. UIMA CPE/ CPM (collection processing engine / collection processing manager), UIMA AS (Asynchronous scale-out), UIMAfit simple runtime, or UIMAfit hadoop runtime, are all one of the UIMA runtimes.

The LAPs of EXCITEMENT exist primarily for processing textual entailment data. With this goal in mind, it is possible to formulate more focused requirements for EXCITEMENT LAP components. An EXCITEMENT LAP should provide the following capabilities:

- Calling a single AE or multiple AEs: Ideally, the user code can utilize AEs for its own goals. This is not limited to process T-H pairs, and users can call the provided AEs for general text analysis. For example, they might need some linguistic analysis to fetch possible text fragments, etc. (Note that providing linguistic analysis components as individual AEs can greatly enhance performance, if a user needs linguistic pre-processings. For example, assume that an EDA needs POS tagging and NER, and that a user must run POS tagging on his data to formalize his problem into a textual entailment problem. The user can choose to continue the analyis pipeline for the EDA, if the pipeline is consist of individual AEs.)

- Calling a prepared pipeline for generating EDA inputs: The user code must be able to generate a CAS that can be fed into EDA interface. Thus the LAP must provide pipelines that are compatible with existing EDAs. These pipelines must generate CAS outputs that are compatible with the EXCITEMENT type system (cf. Section 3.3.4, “Additional Types for Textual Entailment”).

- Processing a collection for EDAs: User code can pre-process a collection of documents to generate a set of enriched documents that each can be fed into EDA for TE decision.

3.2.3. What should be provided by the implementer

The EDA implementer should provide:

- The implementer **must** provide linguistic pipelines that can process T-H pairs and generate proper CAS structure that can be consumed by the EDA. This is specified in Section 3.4.2, “Providing an analysis pipeline”.

- The implementer **should** provide collection processing capability that can enrich a set of documents to produce properly serialized CAS structures that can be consumed by the EDA. This is specified in Section 3.4.3, “Providing collection processing”.

- It is **recommended** that the implementer should provide individual linguistic analysis components as individual AEs. This is specified in Section 3.4.1, “Providing individual analysis engines (AEs)”. In some cases, providing individual engine might not be desirable. Treatment of such cases are described in subsection Section 3.4.4, “Alternative path: translating final pipeline result into common representation”.

- All pipeline implementations **must** utilize the common type system. The common type system is described in Section 3.3, “Type Systems”. Sometimes the EDA implementer **may** need to extend the type system to get the best performance. This is discussed in Section 3.3.5, “Extending existing types ”.

This specification provides no further information on UIMA. Please consult UIMA documentation such as [UIMA-doc] to learn more about UIMA framework and how to build UIMA components.

3.3. Type Systems

The annotation layers of a CAS are represented in terms of UIMA types. To ensure the compatibility of components and their consumers, it is vital to have a common type system. UIMA community

already has well developed type systems that cover generic NLP tools. Two notable systems are DKPro [DKPRO] and ClearTK [CLEARTK]. They cover almost all domains of common NLP tools, including dependency and constituent parsing, POS tagging, NER, coreference resolution, etc. Their type systems are also fairly language-independent. For example, the POS tags of DKPro have been used for taggers of Russian, English, French, and German, and its parse annotations have been used in Spanish, German and English parsers.

For generic linguistic analysis results, we adopt the type system of DKPro. The type system is already proven in many applications as a matured type system. Adopted generic types are described in Section 3.3.3, “Types for generic NLP analysis results”, and their actual type definitions in XML are included in Appendix A, *Type Definition: types for general linguistic analysis*.

This specification also defines additional types that are needed for textual entailment systems. They are including text and hypothesis marking, predicate truth annotation, event annotation, etc. They are described in Section 3.3.4, “Additional Types for Textual Entailment”, and their actual type definitions are included in Appendix B, *Type Definition: types related to TE tasks* .

Once a type is defined and used in implementations, the cost of changing to another type system is prohibitive. Therefore, the choice of types is a serious decision. At the same time, the type system has to allow for a certain amount of flexibility. First, it needs to permit users to extend it without impacting other components that use already existing types. It even permits old components that do not aware of new types to work with the data annotated with components that work with extended types (see Section 3.3.5, “Extending existing types ”). Second, types are generally evolving with the community. In this sense, we must assume that our type system (especially the parts defined specifically for textual entailment problems) may face some changes driven by practical experiences with the EXCITEMENT open platform. In contrast, we assume that the adopted generic NLP types of DKPro are relatively matured system, since it has been around for some years.

3.3.1. Relevant UIMA concepts: Artifact, View, SOFA

"Artifact" is an UIMA term which means the raw input. For example, a document, a webpage, or a RTE test dataset, etc, are all artifacts. For the moment, let's assume that we have a webpage as an artifact, and the goal is to provide various analysis on the text of the web page.

The generic analysis tools (like POS tagger, parser, etc) do not know about HTML tags of the webpage. And it is not desirable to add such specific tag support to generic tools. In UIMA, generic tools can be used without modifications for such cases, by adopting "view"s. For the previous example, a html detagger module can generate a plain text version of the webpage. This can be regarded as a new perspective (view) on the original data (artifact). In UIMA this new view can be introduced into a CAS. Once the view is introduced in the CAS, subsequent analysis like POS tagging and parsing can process the plain text view of the web page without knowing that it is from a webpage.

"Subject OF Analysis" (SOFA) is another UIMA concept. It is referring the the data subject that is associated with a given view. In the website example, it is the plain text version of the web page. One view has one SOFA, and one CAS can have multiple views. Views (and Sofas) can have names that can uniquely identify them among a given CAS. Annotations of CAS generally have a span: beginning position and the end positing of the annotated text. Such spans are expressed as offset of each SOFA. Thus, annotators (individual analysis engines) are annotating SOFAs, instead of artifacts directly.

A pictorial example of CAS, views and annotations are given after introduction of the types for textual entailments, in Figure 4, “Example of an entailment problem in CAS”.

3.3.2. Relevant UIMA default types

The UIMA framework itself defines only a handful of default types. They are including primitive types (like string, numerical values), and basic annotation types.

Primitive types only represent simple values, but user-defined types can express arbitrary data by composing primitive types, and other types. For example, a token type might have a string (primitive type) for lemma, another string for POS, and two numbers for starting position and end position of the

token. Within UIMA CAS, this list of values are called as feature structures. Each feature in the feature structure has a name and a type, thus only a specific type can be pointed by the feature.

UIMA type system is based on single-parent inheritance system. Each type must have a one super-type, and it gets all features from the super-type.

The followings are non-exhaustive list of the default types that are used in the adopted and proposed type system of the specification.

- `uima.cas.String`: one of the primitive types. By primitive types, it means that it is not part of normal type hierarchy. Examples of other primitive types are including `Boolean`, `Integer`, `Double`, etc. String is represented as java string (thus Unicode 16 bit code).
- `uima.cas.top`: This type is the top type of the CAS type hierarchy. In the hierarchy, this type is the only type that do not have any super type. All CAS types are subtype of this type. The type do not any feature.
- `uima.tcas.Annotation`: This type supports basic text annotation ability by providing span: starting and ending of this annotation. It has two features, `begin` and `end`. They are `uima.cas.Integer`, and associated with a character position of a SOFA. Most of the annotations used in the adopted type system is a subtype of this type.
- `uima.tcas.DocumentAnntation`: This is a special type that is used by the UIMA framework. It has a single feature `language`, which is a string indicating the language of the document in the CAS.

Fully detailed information on CAS, and types of default CAS can be found in the CAS reference section of the UIMA reference document [UIMA-CAS].

3.3.3. Types for generic NLP analysis results

As mentioned in previous sections, we adopt DKPro types for "generic" analysis layers. This section is informative and here only to help readers with a general understanding. Normative definitions of the types are included in the Appendix A, *Type Definition: types for general linguistic analysis*.

In this section, the string `DKPRO` is used as a short-form of `de.tudarmstadt.ukp.dkpro.core.api`.

3.3.3.1. Segmentation types

Segmentation represents general segments of document, paragraphs, sentences and tokens. They are defined as extension of UIMA annotation types. Paragraphs and sentences are simple UIMA annotations (`uima.tcas.Annotation`) that mark begin and end. Token annotation holds additional informations like lemma (string), stem (string) and POS (POS type).

See `DKPRO.segmentation.*` types.

3.3.3.2. POS types

POS types are defined as extension of UIMA annotation type (`uima.tcas.Anntation`). It has feature of PosValue (string). This string value holds the unmapped (original) value that is generated by the POS tagger. Common POS types are defined as inherited types of this POS type. They are including PP, PR, PUNK, V, N, etc. To see all common POS types, see `DKPRO.lexmorpth.types.pos` and the inherited types (`DKPRO.lexmorph.types.pos.*`).

Note that POS types have inherited structures --- like NN and NP are subtype of N, etc. Thus, if you access all type.N, you will also get all inherited annotations (like NN, NP), too.

3.3.3.3. Document Metadata

DocumentMetadata holds meta data of the document, including document ID and collection ID. It is a sub type of UIMA Document annotation, which holds language and encoding of the document. See `DKPRO.metadata.type.DocumentMetaData`.

3.3.3.4. Named Entity Recognition

Types related to NER data are defined as subtype of UIMA annotation type. It has string feature named "value" that holds output string of NER annotator. Actual entity types are mapped into subtypes that represents organizations, nationality, person, product, etc. See `DKPRO.ner.type.NamedEntity` and its sub types (`DKPRO.ner.type.NamedEntity.*`).

3.3.3.5. Constituent parse result

Constituent parsing results can be represented with DKPRO.syntax.type.constituent. Constituent type and its inherited subtypes. This type represent a parse node, and holds needed informations like `parent` (single node), `children` (array), and constituentType. `constituentType` field holds the raw data (unmapped value) of the parser output, and the mapped value are represented with types, such as `type.constituent.DT`, `type.constituent.EX`, etc. See `DKPRO.syntax.type.constituent.Constituent` and its sub types (`DKPRO.syntax.type.constituent.*`).

3.3.3.6. Dependency parse result

Dependency parse results are represented with type `DKPRO.syntax.type.dependency.Dependency` and its subtypes. The main type has three features: `Governor` (points a `segmentation.type.Token`), `Dependent` (points a `segmentation.type.Token`), and `DependencyType` (string, holds an unmapped dependency type). Common representation of dependency parse tree are built by subtypes that inherit the type. Each subtype represents mapped dependency, like `type.dependency.INFMOD`, `type.dependency.IOBJ`, etc. To see all mapped common dependency types, see `DKPRO.syntax.type.dependency.*`.

3.3.3.7. Coreference Resolution

Coreferences are represented with `DKPRO.coref.type.CoreferenceLink`. It is an UIMA annotation type, and represents link between two annotations. The type holds `next` feature that points another `type.CoreferenceLink`, and a string feature that holds the `referenceType` (unmapped output). Currently it does not provide common type on coreference type. `CoreferenceLink` represent single link between nodes. They will naturally form a chain, and the start point of such chain is represented by `DKPRO.coref.type.CoreferenceChain`.

3.3.3.8. Semantic Role Labels

This subsection describes types related to represent semantic role labels. It consists of two types, `DKPRO.SemanticRole.Predicate` and `DKPRO.SemanticRole.Argument`.

`DKPRO.SemanticRole.Predicate` is a `uima.tcas.Annotation`. It represents a predicate. It holds the predicate sense as string and links to its arguments (An array of `DKPRO.SemanticRole.Argument`). It has the following features:

- `predicateName` (`uima.cas.String`): This feature represents the name of this predicate. It actually refers to the sense of the predicate in PropBank or FrameNet.
- `arguments` (`uima.cas.FSArray`): This feature is an array of `DKPRO.SemanticRole.Argument`. It holds the predicate's arguments.

`DKPRO.SemanticRole.Argument` is a `uima.tcas.Annotation`. It represents an argument. It has two features; the argument name (type), and a backward reference to the predicate that governs the argument. They are:

- `argumentName` (`uima.cas.String`): This feature represents the name of this argument. It refers to the different types of arguments, like A0, A1, AM-LOC, etc.

- predicates (uima.cas.FSArray): This feature is an array of DKPRO.SemanticRole.Predicate. (Backward) references to the predicates which governs it.

Both annotations are applied to tokens (same span to tokens), and the semantic dependencies are implicitly recorded in the FSArrays. The reason to keep the FSArray for both side is that the redundancy makes the later processing easier (e.g. You can find the corresponding predicates easily from each argument, etc).

3.3.4. Additional Types for Textual Entailment

This section is informative and here only to help readers general understanding of types and their data. Normative definitions of the types are included in the Appendix B, *Type Definition: types related to TE tasks* .

3.3.4.1. Representation of Text and Hypothesis

The basic approach of the platform is to use a single CAS to represent a single entailment problem. The CAS that holds the entailment problem provides two named views. One is the TextView and the other is the HypothesisView. The CAS output of an analysis pipeline must provide two views with these names. Each view holds all data the text and the hypothesis, respectively.

EXCITEMENT.entailment.EntailmentMetadata provides metadata for each entailment problem. It is an extension of uima.tcas.DocumentAnnotation, and inherits the language identification (feature language, a uima.cas.String). It also holds additional metadata of the entailment problem:

- task (uima.cas.String): this string holds task description which can be observed in the RTE challenge data.
- channel (uima.cas.String): this metadata field can holds a string that shows the channel where this problem was originated. For example, "customer e-mail", "online forum", or " customer transcription", etc.
- origin (uima.cas.String): this metadata field can hold a string that shows the origin of this text and hypothesis. A company name, or a product name.

Text-Hypothesis pairs are represented by the type EXCITEMENT.entailment.Pair. The type is a subtype of uima.tcas.Annotation, adding references to text and hypothesis. If the CAS represents multiple text and/or hypothesis, the CAS will holds multiple Pair instances. The type has following features:

- pairID (uima.cas.String): ID of this pair. The main purpose of this value is to distinguish a certain pair among multiple pairs.
- text (EXCITEMENT.entailment.Text): this features points the text part of this pair.
- hypothesis (EXCITEMENT.entailment.Hypothesis): this features points the hypothesis part of this pair.
- goldAnswer (EXCITEMENT.entailment.Decision): this features records the gold standard answer for this pair. If the pair (and CAS) represents a training data, this value will be filled in with the gold standard answer. If it is a null value, the pair represents a entailment problem that is yet to be answered.

EXCITEMENT.entailment.Text is an annotation (extending uima.tcas.annotation) that annotates a text item within the TextView. It can occur multiple times (for multi-text problems). It has no feature. A Text instance is always referred to by a Pair instance.

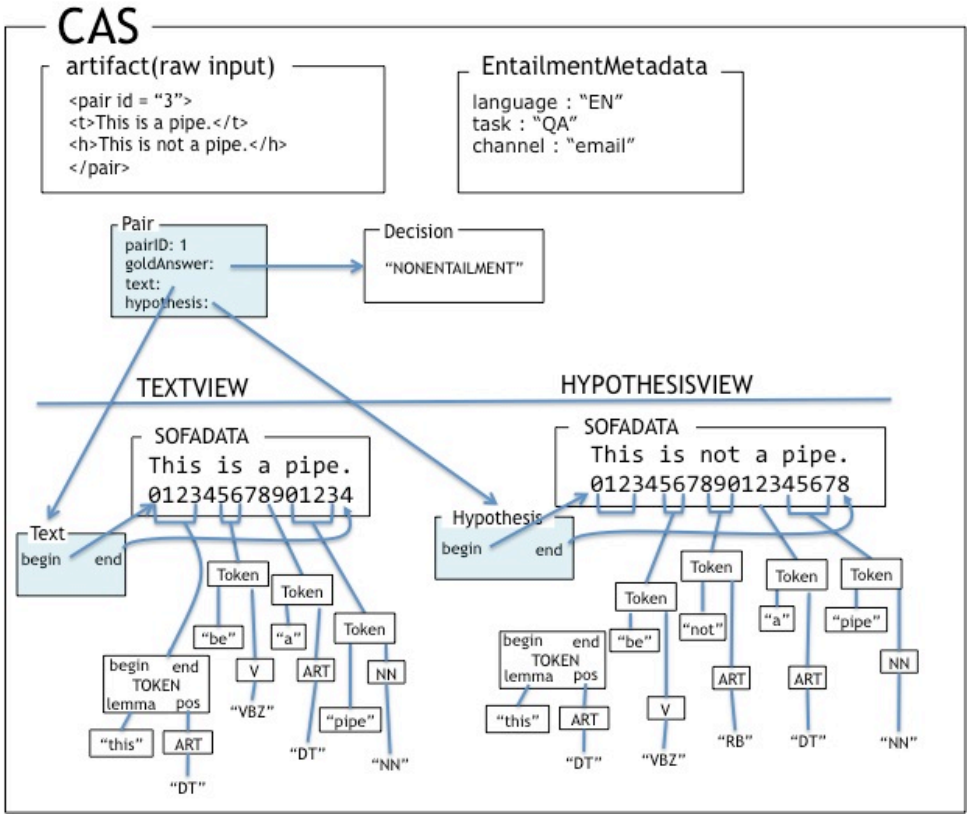
EXCITEMENT.entailment.Hypothesis is also an annotation (uima.tcas.annotation) that annotates a hypothesis item within the HypothesisView. It can occur multiple times (in multiple-hypothesis cases). It has no feature. A Hypothesis instance is always referred to by a Pair instance.

The EDA `process` interface receives CAS data as a JCas object (the UIMA-created Java type hierarchy corresponding to the UIMA CAS types). CASes that feed into the EDA `process` interface must have at least a single `entailment.Pair`. Naturally, each view (`TextView` and `HypothesisView`) must have at least one `entailment.Text` and `entailment.Hypothesis`. Multiple text and multiple hypothesis cases will be represented by multiple number of `entailment.Pair`. Note that the relationship between pairs on one side and texts and hypotheses is not a one-to-one relationship. Several pairs can point the same hypothesis, or to the same text. For example, if one hypothesis is paired with several potential texts, there will be multiple pairs. All pairs will point to the same hypothesis, but to different texts.

`EXCITEMENT.entailment.Decision` represents the entailment decision. It is a string subtype, which is a string (`uima.tcas.String`) that is only permitted with predefined values. `entailment.Decision` type can only haves one of "ENTAILMENT", "NONENTAILMENT", "PARAPHRASE", "CONTRADICTION", and "UNKNOWN" (The type can be further expanded in the future).

The following figure shows a pictorial example of an entailment problem represented with the types in a CAS. In this example, the original input was a very simple XML annotation of a T-H pair. This input is shown in the artifact (UIMA term of raw input). From this input, two "plain text" analysis subject is fetched. They are listed in SOFA-data of each view. Each view is enriched by token annotations and POS annotations. All annotation types (sub types of `uima.tcas.annotation`) have `begin` and `end` feature, which shows the span of the annotation. (POS types also has the begin and end span, however, they are omitted here for clarity.)

Figure 4. Example of an entailment problem in CAS



The figure shows a single `entailment.Pair`, which points `entailment.Text` and `entailment.Hypothesis`. The `Pair` has an ID, and it also has a `goldAnswer`. The CAS has no additional pair, and the entailment problem described in this example is a single T-H pair. There is one `entailment.EntailmentMetadata` in this CAS. It shows that the language of the two views is "EN" (English as defined in [ISO 639-1]), and the task is "QA". It also has a string for `channel` value as "email".

Figure 5. Example of an entailment problem with multiple pairs

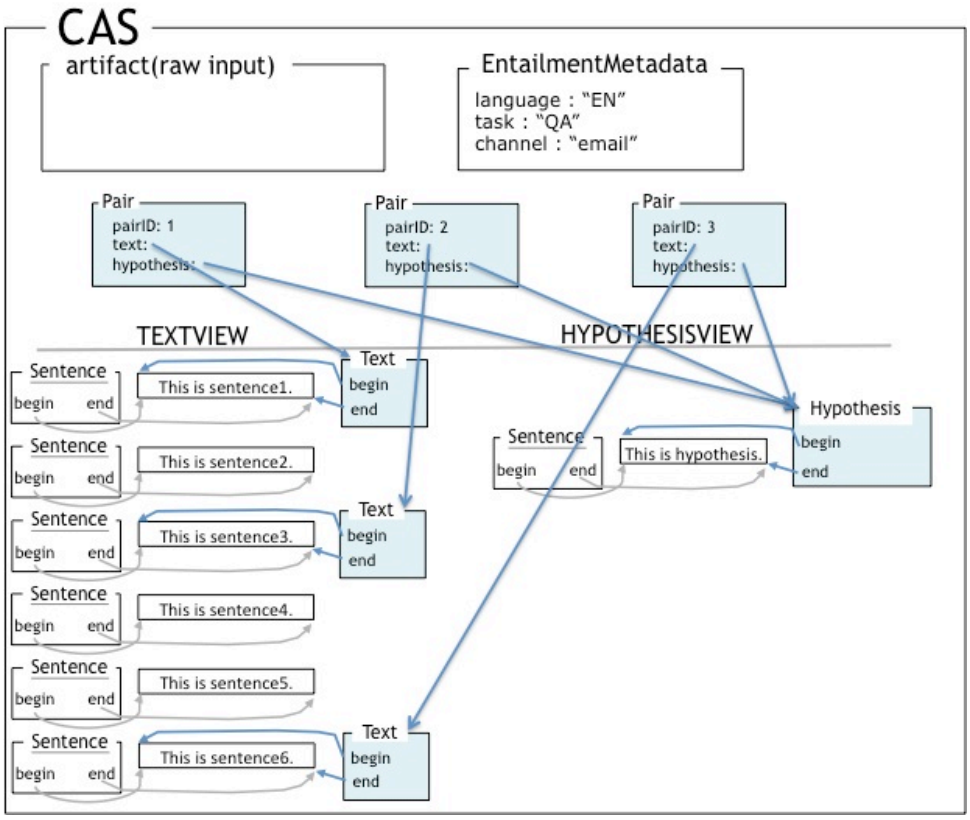


Figure 5, "Example of an entailment problem with multiple pairs" shows another example of the `entailment.Pair`. In this figure, the CAS has multiple pairs. The TextView holds a document with six sentences. The Text annotation marks first, third and sixth sentences. The HypothesisView holds only a single sentence, which is the single Hypothesis annotation. The CAS holds three pairs, and it is a multiple text entailment decision problem. In the figure, linguistic analysis annotations other than sentences are omitted for clarity. The sentences that are not annotated with a Text annotation can be treated as context for the annotated texts. For example, co-reference resolutions will take benefits from the sentences. Also, components like distance calculation components, or text rewriting components can use the terms in context sentences to get better results. Note that EDAs only decide entailment of pairs that are explicitly marked with `Pair` annotations. Existence of text or hypothesis annotation does not automatically make them a candidate. For example, if the pair 2 is missing from the figure, the `Text` annotated on sentence3 will not be compared to the hypothesis sentence.

3.3.4.2. Types for Predicate Truth

This subsection describes types related to the representation of predicate truth values. Predicate truth annotation is an annotation that will provides truth value for all predicates and clauses by traversing the parse tree and take into account implications, negations, and clausal embeddings [PredicateTruth].

We need four annotation types according to the annotations that are needed for predicate truth annotator. They are `EXCITEMENT.PredicateTruth.PredicateTruth`, `EXCITEMENT.PredicateTruth.ClauseTruth`, `EXCITEMENT.PredicateTruth.NegationAndUncertainty`, and `EXCITEMENT.PredicateTruth.ImplicationSignature`. They add annotations to tokens. `ClauseTruth` can annotate a set of consecutive tokens, while other types only annotate a single token.

`EXCITEMENT.PredicateTruth.PredicateTruth` is a `uima.tcas.Annotation`. It represents a predicate truth value annotation. It has the following feature:

- `value` (`PredicateTruthValue`, a string subtype): This represents the value of the annotation. The value type subtype of string permitting only the values `PT+`, `PT-`, and `PT?`.

EXCITEMENT.PredicateTruth.ClauseTruth is a uima.tcas.Annotation. It represents a clause truth value annotation. Note that the begin and end of this annotation can span more than a token. It has the following feature:

- value (ClauseTruthValue, a string subtype): This represents the value of the annotation. The value type is a subtype of string permitting only the values CT+, CT−, and CT?.

EXCITEMENT.PredicateTruth.NegationAndUncertainty is a uima.tcas.Annotation. It represents a negation-and-uncertainty annotation. It has the following feature:

- value (NegationAndUncertaintyValue, a string subtype): This represents the value of the annotation. The value type is a subtype of string permitting only the values NU+, NU−, and NU?.

EXCITEMENT.PredicateTruth.ImplicationSignature is a uima.tcas.Annotation. It represents an implication signature. It has the following feature:

- value (ImplicationSignatureValue, a string subtype): This represents the value of the annotation. The value type is a subtype of string permitting only the values "+/-", "+/?", "?/-", "-/+", "-/?", "?/+", "+/+", "-/-", "?/?".

3.3.4.3. Types for Temporal Expression

The temporal expression types are needed for NER based event models, and they are used in event-based EDAs. This section defines two related types. One is EXCITEMENT.temporal.TemporalExpression and the other is EXCITEMENT.temporal.DefaultTimeOfText.

EXCITEMENT.temporal.DefaultTimeOfText is a uima.tcas.Annotation. It is anchored to a textual region (a paragraph, or a document), and holds the "default time" that has been determined for this passage and can be useful to interpret relative time expressions ("now", "yesterday") in the text. It has one feature:

- time (uima.cas.String): This feature holds the default time for the textual unit which is annotated by this annotation. The time string is expressed in the normalized ISO 8601 format (more specifically, it is a concatenation of the ISO 8601 calendar date and extended time: "YYYY-MM-DD hh:mm:ss").

The second type, EXCITEMENT.temporal.TemporalExpression annotates a temporal expression. It is a uima.tcas.Annotation that annotates a temporal expression within a passage, adding a normalized time representation. It holds two string features: One contains the original temporal expression, and the other contains a normalized time representation, using ISO 8601 as above.

- text (uima.cas.String): This feature holds the original expression appeared on the text.

- resolvedTime (uima.cas.String): This feature holds the resolved time in ISO 8601 format. For example, "Yesterday", will be resolved into "2012-11-01", etc. See the type DefaultTimeOfText for details.

3.3.4.4. Types for Text Alignment

A simple annotation type is provided for text alignment. EXCITEMENT.alignment.AlignedText represent an aligned textual unit. It is a uima.tcas.Annotation. Its span refers to the "source" linguistic entity. This can be a token (word alignment), a syntax node (phrase alignments), or a sentence (sentence alignment). It has a single feature that can point other instances of of AlignedText:

- alignedTo (uima.cas.FSArray of type EXCITEMENT.alignment.AlignedText): This feature holds references to other AlignedText instances. Note that this array can have null, one, or multiple AlignedText.

alignedTo feature can have multiple references, which means that it is one-to-many alignment. Likewise, a null array can also be a valid value for this feature, if the underlying alignment method is

an asymmetric one; empty array means that this `AlignedText` instance is a recipient, but it does not align itself to other text.

3.3.5. Extending existing types

It is possible that users want to add additional information to entailment problems. For example, task-oriented TE modules might want to include additional information such as ranks among texts, relevance of each hypothesis to some topics, etc.

The canonical way to represent such information is to extend the `entailment.Pair` and related types. The type and related types (like `MetaData`, `Text` and `Hypothesis`, etc) are presented to serve the basic need of the EXCITEMENT platform, and additional data can be embedded into CAS structure by extending the basic types.

Naturally, the extension should be performed in a *consistent* manner. This means that the implementer can only define a new type that is extending the existing types, and may not change already existing types. Also, it is recommended that attributes and methods inherited from the existing types should be used in the same way so that components and EDAs which are unaware of the extensions can still can operate on the data.

The type `EXCITEMENT.entailment.Decision` is also open for future extensions. When we define a new relation between text and hypothesis, this type should be first extended to cover the new relation. (This extension should always done with the extension of internal `DecisionLabel` enumeration, see Section 4.2.1.6, “enum `DecisionLabel` ”.

Unlike `EXCITEMENT` types, generic types (of Section 3.3.3, “Types for generic NLP analysis results”) should in general not be extended. Exceptional cases may arise, though, for example when an EDA utilizes some additional information of a specific linguistic analysis tool. In such a case, one may choose to extend generic types to define special types (for example, defining myNN by extending NN, etc). The EDA can then confirm the data is processed by a specific tool (like existence of myNN) and can use the additional data that is only available in the extended type. Other modules that do not recognize the extended types are still able to use the super-types in the output (in the example, myNN will map onto NN).

3.4. Providing analysis components for LAP

3.4.1. Providing individual analysis engines (AEs)

It is recommended that each linguistic analysis module (like POS taggers, parsers or coreference resolvers) should be provided as a generic individual UIMA analysis engine (AE). Thus, user code should be able to call these modules as UIMA components, as described in Section 3.2.2, “Capabilities provided by the EXCITEMENT linguistic analysis pipeline”.

By generic, we mean they are supposed to process normal text and not T/H pairs. The modules are not supposed to process or even be aware of annotations related to entailment problems. This ensures their reusability. The necessary specific processing for entailment data (H/T markup) should be implemented with a complete pipeline, cf. the following subsection.

As described in Section 3.3.5, “Extending existing types ”, all analysis engines must annotate the given data with the common type system. In the rare case where the provided common type system is not sufficient, the implementer must not modify any of the existing types and should only expand existing types with an own type that inherits existing types.

It is recommended that the implementer should reuse existing AEs as much as possible. For example, DKPro already provides many of the well known NLP tools including major parsers and taggers for various languages. ClearTK or OpenNLP also ships various NLP tools as UIMA components. ClearTK or OpenNLP uses different type systems, thus one needs to add additional glue code to map the types into adopted DKPro types. However, such mappings are generally trivial and should not pose large problems.

Note that there is more than one flavor in providing UIMA components. This specification only defines types and components in terms of the bare UIMA framework. Each component is supposed to have its component description, and should be runnable by the UIMA framework. However, the implementers are free to utilize additional tools. For example, DKPro and ClearTK both uses UIMAFit as tool layer, which provides the functionalities of the UIMA framework with additional tool layers. The implementer may utilize such tools, as long as the end results are compatible with what are defined by this specification.

When an implementer includes a third party analysis tool into the LAP, they should check the license of the importing tool and make sure that the license is compatible with the EXCITEMENT platform license. Note that not all open source license are mutually compatible -- for example, some commercial friendly open source license is not compatible with GPL, etc.

In some cases, one may choose NOT to provide individual analysis components as UIMA components. See Section 3.4.4, “Alternative path: translating final pipeline result into common representation” for such as case.

3.4.2. Providing an analysis pipeline

Individual analysis engines (AEs) are generic. They process a single input (single CAS) and add some annotation layers back to CAS. As mentioned above, however, EDAs require more the sum of single AE annotations, namely T-H views, and T-H pair annotations. Thus, it is desirable to provide complete "pipelines" that is properly annotated with types needed for textual entailment.

A pipeline for EXCITEMENT EDA must process the followings:

- The end result of the pipeline should generate the CAS as an entailment test case, as described in Section 3.3.4, “Additional Types for Textual Entailment”.
- It should be able to process input of formatted text that follows the EXCITEMENT test data format, defined in Section 5.2, “Input file format ”.
- All linguistic analysis annotations should follow the adopted type system (of Section 3.3.3, “Types for generic NLP analysis results”), or some extension of the adopted types.

EDA implementers **must** provide at least a single pipeline that produces suitable input for their EDA, or declare that an already existing pipeline as the suitable pipeline. If the EDA supports additional input mode like multiple texts/hypotheses, the implementer should provide a corresponding pipeline that produce multiple text/hypothesis.

3.4.3. Providing collection processing

Collection processing is often needed for NLP applications. The implementer can provide collection processing with a UIMA runtime, like CPE/CPM. In an actual implementation, the collection processing can be achieved by adding a collection reader and an CAS serializer to an AAE, and by running it on the UIMA runtime.

EDA implementers **should** provide the collection processing capability with similar capabilities to pipelines:

- It must process a collection of documents stored in a path.
- The end result for each file process must generate the CAS as an entailment test case, as described in Section 3.3.4, “Additional Types for Textual Entailment”. The CAS should be stored in a path as the XMI serialization.
- Each stored CAS data must be identical to the output of the corresponding pipeline.
- It should be able to process input of formatted text that follows the EXCITEMENT test data format, defined in Section 5.2, “Input file format ”.

- All linguistic analysis annotations should follow the adopted type system (of Section 3.3.3, “Types for generic NLP analysis results”), or some extension of the adopted types.

Note that XMI (XML Media Interchange format) serialization is the standard serialization supported by UIMA framework. It stores each CAS as an XML file, and the stored files can be easily deserialized by UIMA framework utilities. You can find additional information about CAS serialization in [UIMA-ser]

3.4.4. Alternative path: translating final pipeline result into common representation

It is recommended that individual analysis components are supplied as UIMA analysis engine (AE). This will make sure those analysis modules can be used by top level users in various situations, as suited for the need of application programmers.

However, there can be extreme cases where EDAs are depending on specialized analysis modules forming a pipeline that cannot easily be divided into individual UIMA analysis engines. For example, assume that there is a parser and a coreference resolution resolver that are designed to work together to produce a specific output for a given EDA. Then, it might not be desirable to break them into individual analysis engines. In such a case, one may choose to provide only the pipeline as a single analysis engine which translates the final output of the pipeline into the common representation format.

Even in this cases, the developer must provide the pipeline as a UIMA component. This will ensure that the pipeline can be called in an identical manner to normal pipelines.

4. Common Interfaces of the Entailment Core

This section defines the interfaces and types of the Entailment Core. Section 4.1 provides a list of requirements and some methodological requirements. The remaining subsections contain the actual specification. This section is still subject to update, following the ongoing discussions of the WP3 interface definition subgroup. We solicit comments.

4.1. Requirements for the Entailment Core

4.1.1. Requirements for the EDA

- *Entailment recognition (classification mode).* The basic functionality that EDAs have is to take an unlabeled text-hypothesis pair as input and return an entailment decision.
- *Entailment recognition (training mode).* Virtually all EDAs will have a training component which optimizes its parameters on a labeled set of text-hypothesis pairs.
- *Confidence output.* EDA should be able to express their confidence in their decision.
- *Support for additional modes.* The platform should support the processing of multiple Hs for one T, multiple Ts for one H, as well as multiple Ts and Hs.

4.1.2. Requirements for the Components

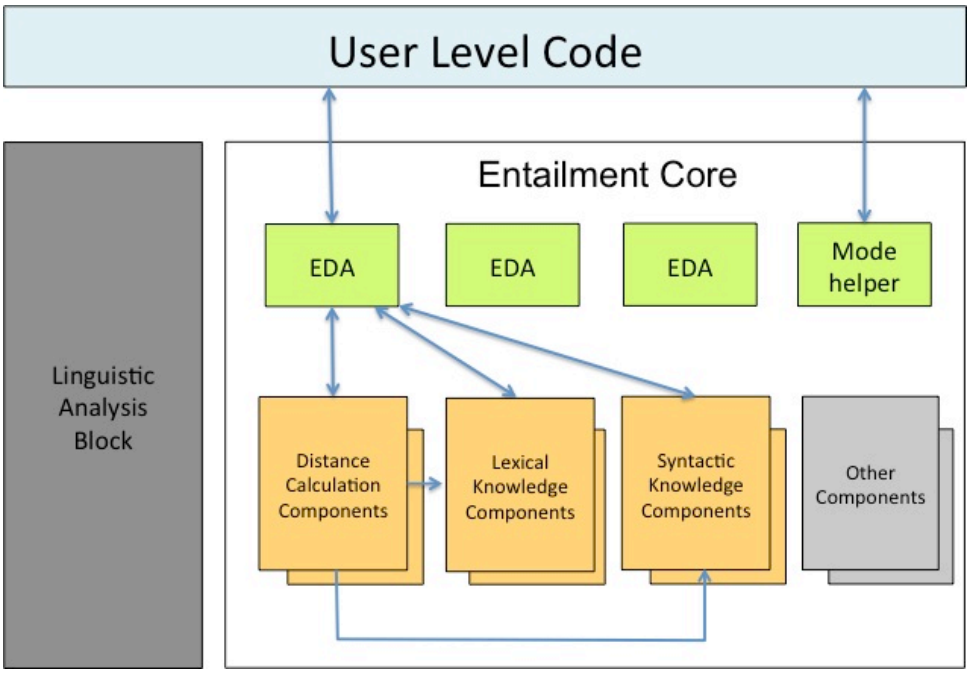
- *A small set of reusable component classes.* The specification should describe a small set of component classes that provide similar functionality and that are sufficiently general to be reusable by a range of EDAs.
- *Interfaces and types for component classes.* The specification should specify user interfaces for these component classes, including the types used for the exchange of information between components and EDA.
- *Further extensibility.*Users should be able to extend the set of components beyond the standard classes in this specification. This may however involve more effort.

To determine the set of reusable components and their interfaces, we started from the three systems that we primarily build on and followed a *bottom-up generalization* approach to identify shared functions (components), interfaces and types. The results are discussed in Section 4.1.3, “Identifying common components and interfaces ”. Further functionality of these components (e.g. their training, if relevant) will not be specified.

4.1.3. Identifying common components and interfaces

By following the bottom-up approach, we have identified a set of major common interfaces. The following figure shows the common components within the entailment core.

Figure 6. Common components within EXCITEMENT entailment core.



The top-level interfaces of the entailment core are consist of EDA interface and mode helper interface. EDAs are the main access point of entailment decision engine. They get annotated input in the form of CAS, and returns a decision object. The EDA interface also includes support for multiple Text and/or multiple Hypothesis modes. The EDA interfaces are described in Section 4.2, “EDA Interface ”. The mode helper is a wrapper tool that supports multiple text and multiple hypothesis mode for EDAs that can only process single T-H pairs. Its interface is described in Section 4.3, “Mode Helper”.

We have identified three main classes of components. **Distance calculation** components are one type of common components that is important for many entailment engines. EDAs can use the distance between T-H pairs as primary decision factor, or can use them as features for decision algorithms. The interface of distance calculation components specifies that they accept a CAS as the input, and return an object the distance between the two views of the CAS. The interface is described in Section 4.5, “Interface of distance calculation components ”.

We also define two component classes that deal with different kinds of linguistic knowledge, namely lexical knowledge and syntactic knowledge. **Lexical knowledge** components describe relationships between two words. Various lexical resources are generalized into EXCITEMENT common lexical knowledge interface. The interface is described in Section 4.6, “Interface of lexical knowledge components ”. **Syntactic knowledge** components contain describes entailment rules between two (typically lexicalized) syntactic structures, like partial parse trees. For entailments, the knowledge can be generalized into rewriting rules at partial parse trees, where left hand sides are entailing the right hand sides. The common interface for syntactic knowledge is described in Section 4.7, “Interface of syntactic knowledge components ”.

4.1.4. Using CAS or independent types in the Entailment Core?

The EXCITEMENT platform uses UIMA CAS type system for the linguistic analysis pipeline (Section 3). The UIMA types are strong enough to describe various data structure, and work well as in-memory and serialized common representation. It is thus a natural question whether CAS types can be re-used for the communication between EDAs and components.

Our general guideline within the Entailment Core is that the interfaces are supposed to use UIMA CAS only when a complete text (Text or Hypothesis, potentially with context) is exchanged as a parameter. The reason is that even though CAS types (JCas objects) are expressive enough to describe any data, they are basically flat structures --- tables with references. This limits their capability to be used

efficiently. A second reason is that JCas objects only works within CAS context. Even simple JCas objects like tokens need a external CAS to be initialized and used. This makes it impractical to use JCas objects and their types in interfaces that do not refer to complete texts (which are modeled in CASes). To repeat, we use JCas and related types only when a reference to the whole text is exchanged. Furthermore, we only exchange references to the top-level JCas object and never to any embedded types like views or annotation layers.

A second aspect that must be handled with care is the extension of JCas objects. JCas provides the flexibility to extend objects while keeping the resulting objects consistent with the original JCas types. However, this can cause duplicated definitions when mismatches are introduced. Moreover, CAS types will be used and shared among all EXCITEMENT developers which makes extensions introduced by one site problematic. For these reasons, our policy is that JCas objects should not be extended.

4.2. EDA Interface

The common interface of EDA is defined with two types of Java Interface. `EDABasic` interface defines methods related to single T-H pair process and training. `EDAMulti*` interfaces defines additional optional interfaces for multiple text and/or hypothesis cases.

4.2.1. EDA Basic interface: interface `EDABasic`

This interface defines the basic capability of EDAs that all EDAs must support. It has four methods. `process` is main access method of textual entailment decision. It uses `JCas` object as input, and `TEDecision` interface as output. The interface as Java code is listed in Section C.1, “`interface EDABasic` and related objects”.

4.2.1.1. `initialize` method

```
public void initialize(CommonConfig config)
```

This method will be called by the top level programs as the signal for initializing the EDA. All initialization of an EDA like setting up sub components and connecting resources must be done in this method. An EDA implementation must check the configuration and raise exceptions if the provided configuration is not compatible.

`initialize` is also responsible for passing the configuration to common sub-components. At the initialization of core components (like distance calculation components or knowledge resource components), they will raise exceptions if the configuration is not compatible with the component. `initialize` must pass through all exceptions that occurred in the subcomponent initialization to the top level.

Recommended ways of meta data checking policy of EDA and the core components are described with more detail in Section 4.8.1, “Recommended policy on metadata check”.

- Parameters
  - `config`: a common configuration object. This configuration object holds platform-wide configuration. An EDA should process the object to retrieve relevant configuration values for the EDA. Common configuration format and its in-memory object is defined in section Section 5.1, “Common Configuration ”.

4.2.1.2. `process` method

```
public TEDecision process(JCas aCas)
```

This is the main access point for the top level. The top level application can only use this method when the EDA is properly configured and initialized. Each time this method is called, the EDA should check the input for its compatibility. Within the EXCITEMENT platform, EDA implementations are decoupled with linguistic analysis pipelines, and you cannot blindly assume that CAS input is valid. EDA implementation must check the existence of proper annotation layers corresponding to the configuration of the EDA.

The TE decision is returned as an object that implements `TEDecision` interface which essentially holds the decision as enum value, numeric confidence value and the placeholder for additional info.

- Parameters
  - `aCas`: a JCas object. Which has two named views (TextView and HypothesisView). It has linguistic analysis result with `entailment.Pair` as defined in Section 3.3.4, "Additional Types for Textual Entailment".
- Returns
  - `TEDecision`: An object that follows `TEDecision` interface. See Section 4.2.1.5, "`TEDecision` interface ".

4.2.1.3. `shutdown` method

```
public void shutdown()
```

This method provides a graceful exit for the EDA. This method will be called by top-level as a signal to disengage all resources. Resources hold by the `initialize` should be released after this method is called.

4.2.1.4. `start_training` method

```
public void start_training(CommonConfig c)
```

`start_training` interface is the common interface for EDA training. The interface signals the start of the training with the given configuration `c`.

If the EDA is trainable, the EDA implementation should provide the following capability:

- (Mandatory: Single T-H cases) It must be able to train itself on a collection of serialized CAS files, where each CAS represents an `ExcitementTrainingPair`.
- (Optional: Multiple T-H cases) It may support the capability to train on a collection of serialized CAS files, where each CAS represents an `ExcitementTrainingData`.
- It may provide support of any additional training data type.
- It should provide all relevant parameters related to the training in the common configuration.
- Upon a successful training, it should result one or more model file, in a path described in the configuration. And the users should be able to use the model file(s) on future `process()` by passing the model via configuration, on `initialize()`.

`start_training` and the consequent training process should do the capability check (with the configuration), and compatibility check (with the training data). If there is some incompatibility, the component should raise exceptions accordingly.

4.2.1.5. `TEDecision` interface

This interface represents the return value for `process` interfaces.

- Methods
  - `DecisionLabel getDecision()`: this method returns an enumeration type `DecisionLabel` as the entailment decision.
  - `double getConfidence()`: this method returns the associated confidence value for the entailment decision. The range is [0,1], and 1 means full confidence. If the value is not meaningful for the EDA, it should return a constant number `CONFIDENCE_NOT_AVAILABLE`, which is defined in the interface as a constant.
  - `String getPairID()`: this method returns the `entailment.Pair` id as described in the CAS.

- `Object getInfo()`: This method returns an arbitrary object, which is optional and should hold additional information about the decision process. If no additional information is available, it should return `null`. If it returns a non-null object, the type and usage of the object should be documented with the EDA. Also `toString` of the object should output some meaningful text.

4.2.1.6. enum `DecisionLabel`

This enumeration value represents the entailment decision. It should be implemented as a hierarchical enumeration. One possible hierarchical enumeration with method `is()` is listed in Section C.1.3, "`enum DecisionLabel`" The specification 1.0 defines 6 labels: `Entailment`, `NonEntailment`, `Contradiction`, `Paraphrase`, `Unknown`, and `Abstain`. Note that the enum values have hierarchy, and related. On top, there is entailment and non-entailment. For example, `Paraphrase` is an `Entailment` (which means that `DecisionLabel.Paraphrase.is(DecisionLabel.Entailment)` will return true.). Likewise, `Contradiction` and `Unknown` are also `NonEntailment`.

This type is open for extension. With the hierarchical enumeration, one can extend this enumeration type with backward compatibility. Note that the extension should always done with UIMA type extension of `EXCITEMENT.entailment.Decision` Section 3.3.4, "Additional Types for Textual Entailment").

Usage of `Abstain` should be minimized. `Abstain` values should not be used in the top-level independent EDAs. They should be only used in special cases, like filtering type EDAs of TIE.

4.2.2. EDA multiple text/hypothesis interface: interface `EDAMultiT`, `EDAMultiH`, `EDAMultiTH`

Each of the interface defines a method, namely `processMultiT`, `processMultiH`, and `processMultiTH`. EDA may support them: supporting multiple T/H interfaces are optional for an EDA. Multiple Texts and Hypotheses are marked in the CAS by multiple `EXCITEMENT.entailment.Pair` annotations.

4.2.2.1. `processMultiT`, `processMultiH`, `processMultiTH`

```
public List<TEDecision> processMultiT(JCas aCas)

public List<TEDecision> processMultiH(JCas aCas)

public List<TEDecision> processMultiTH(JCas aCas)
```

The processMulti methods share the same signature. They all work on a single CAS that holds multiple texts and/or hypotheses, and returns a list of Decision objects. Each `TEDecision` object has its own pair ID (`getPairID()`) --- as annotated in `entailment.Pair` data. Just like single T-H `process` interface, `processMulti*` should check input CAS, and have to raise proper exceptions if some needed CAS structure is missing.

The specification do not define the ordering of the resulting list.

- Parameters
  - `aCas`: JCas object with `TETextITem` and/or `TeHypothesisItem` annotations.
- Returns
  - `List<TEDecision>`: A list of `TEDecision` objects.

4.3. Mode Helper

The "mode helper" is a wrapper implementation that uses a single-mode EDA (`EDABasic`) to support multiple T-H cases (that of `EDAMultiT`, `EDAMultiH`, and `EDAMultiTH`). It receives an already

initialized EDA and iteratively calls the EDA to produce results for multiple T/H cases. It is a baseline implementation that can be used with any EDA.

It is defined as an abstract java object, and it is expected that the platform will ship with a simple, non-optimized implementation. The Java code is listed in Section C.3, "`class ModeHelper`" It is expected that the platform will share one implementation of mode helper for all EDAs.

4.3.1. Methods

4.3.1.1. `setEDA` method

`public void setEDA(EDABasic eda)` This method must be called before calling any of the process methods. Note that initialization and shutdown of the EDA must be effected by user code, outside the mode helper.

- Parameters
  - `eda`: An initialized `EDABasic`. All processing of the mode helper will be done by calling `eda.process()`.

4.3.1.2. `processMulti` methods

```
public List<TEDecision> processMultiT(JCas aCas)

public List<TEDecision> processMultiH(JCas aCas)
```

Mode helper implements two `EDAMulti*` interfaces. When called, they will iteratively call `process()` of the given EDA to generate the result. All other behaviors (input specifications and exception checks, etc) should be the same with `EDAMulti*` interface that implements the methods directly.

4.4. Common functionality of components: The `Components` interface

There is a small set of methods that are common to all components whatever their internal structure. These methods are primarily concerned with the adminstrative functions and the interaction with the EDA. These methods form the `Components` interface. All more specific interfaces as outlined in the following sections are subinterfaces of `Components`.

4.4.1. method `initialize()`

```
public void initialize(CommonConfig config)
```

This method will be called by the component user as the signal for initializing the component. All initialization (including connecting and preparing resources) should be done within this method. Implementations must check the configuration and raise exceptions if the provided configuration is not compatible with the implementation.

- Parameter `config`: a common configuration object. This configuration object holds the platform-wide configuration. An implementation should process the object to retrieve relevant configuration values for the component. The common configuration is defined in Section 5.1, "Common Configuration ".

4.4.2. `getComponentName` method

```
public String getComponentName()
```

This method provides the (human-readable) name of the component. It is used to identify the relevant section in the common configuration for the current component. See Sections Section 5.1.2, "Overview of the common configuration " and Section 4.8.3, "Component name and instance name".

4.4.3. `getInstanceName` method

```
public String getInstanceName()
```

This method provides the (human-readable) name of the instance. It is used to identify the relevant subsection in the common configuration for the current component. See Sections Section 5.1.2, “Overview of the common configuration ” and Section 4.8.3, “Component name and instance name”.

4.4.4. `Storage of names`

The specification does not define how and where component and instance names is stored internally within the component implementations. It is expected that the implementation efforts will implement a common practice of keeping instance names. In any case, components must provide extensive documentation about the component and instance names that they assume.

4.5. Interface of distance calculation components

The capability to calculate distance or similarity of two given textual units are essential in many textual entailment algorithms. Distance calculations and similarity calculations are generalized with the distance calculation interface within the EXCITEMENT platform. The interface generalizes both similarity and distance calculations in a normalized range of distance 0 (totally identical) to distance 1 (maximally different).

4.5.1. interface `DistanceCalculation`

The interface is a subinterface of `Component`. The new methods it adds is `calculation()`, which gets a `JCas` and returns an object of `DistanceValue`. The interface definition in Java code can be found in Section C.4, “`interface DistanceCalculation` and related objects”.

4.5.1.1. method `calculation()`

```
public DistanceValue calculation(JCas aCas)
```

This method must be called only after a successful call to `initialize()`. It delivers the distance calculation result in an Object `DistanceValue`, which represents the distance between two textual objects in the JCas. The calculation is done between the two views of the JCas: TextView and HypothesisView. Note that the view names are irrelevant to the actual service provided by this interface. More appropriate names like "LeftHandSideView" and "RightHandSideView" are not defined just to simplify the data access. `calculation()` method knows nothing about other entailment annotations (like `TextItem` or `HypothesisItem`) and it should not check those annotations.

The implementation may check the validity of the input. For example, if the provided annotations are not compatible for calculations (like missing parse tree, for tree-edit-distance), it can raise an exception accordingly. However, (unlike EDA `process()` method) this check is not mandatory.

4.5.2. Type `DistanceValue`

This Java type that holds distance calculation result. It has a few member variables and public access functions for the variable. The type is used to exchange data between the component and the EDA and is thus immutable. Its variables are set during initialization. The actual java code listing is included in Section C.4, “`interface DistanceCalculation` and related objects”.

4.5.2.1. Variables and access methods

- `private double distance`: The normalized distance. Maximum value is 1 (maximally different), and minimum value is 0 (totally identical). Access is provided by the method `double getDistance()`.

- `private boolean isSimBased`: This boolean is `true` if the calculation is based on similarity functions. This boolean is `false` (default) if the calculation is based on distance-based calculations. This value is provided to help the interpretation of the `unnormalizedValue`. Users can ignore this value, if the do not use `unnormalizedValue`. The method `getIsSimBased()` accesses this value.

- `private double unnormalizedValue`: This variable holds a distance or similarity value that is not normalized. If the value is mapped into the range with common normalization, it will produce the value stored at `distance`. This unnormalized value is provided for the users to use some other methods of normalizations, such as asymmetric normalizations used in some EDAs. The value is accessed by the function `double getUnnormalizedValue()`.

- `private Vector distanceVector`: This variable holds a set of double values. The vector is an optional value that permits the distance calculation components to return a set of distance values that is needed, or used to generate the main value. If the component does not provide this vector, this variable should be `null`.

Note that the value `unnormalizedValue` is not only unnormalized, but also not mapped into the distance scales. Thus, the interpretation of the unnormalized value should always consult `isSimBased` bool value. For example, `unnormalizedValue` of cosine similarity will hold original vector product value, with `isSimBase` as `true`. In this case, higher `unnormalizedValue` means more similar and less distant objects.

4.6. Interface of lexical knowledge components

The access to lexical knowledge is generalized with common interfaces and common types. Each entry of lexical knowledge is encapsulated by `class LexicalRule`, and collections of such rules are represented by the interface `LexicalResource`.

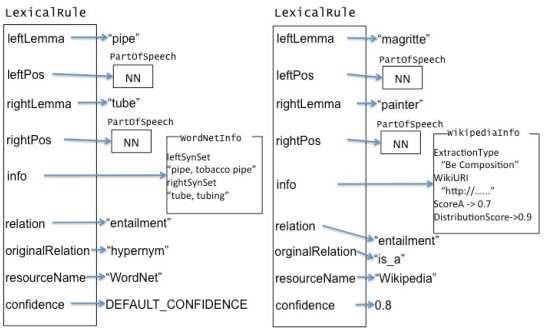
4.6.1. Type `LexicalRule`

```
public final class LexicalRule<I extends RuleInfo>
```

This type represents a generalization of lexical relationships between two words, which are provided by resources like WordNet, VerbOcean and distributional resources. It has two parts: a left hand side and a right hand side. The basic arrangement of lexical resource is that the left hand side entails the right hand side. Additional pieces of information include confidence, relation, and resource name. Finally, lexical rules are parametrized by a type `I` which allows the type to hold additional resource-specific information.

Before going into details of the class variables, let's first see a few examples. The following figure shows a pictorial example of the data type. It shows two instances of lexical knowledge. Each knowledge is represented with a few data features.

Figure 7. Examples of lexical rule instance



On the left, it shows an instance of `LexicalRule` that represents the relationship between "pipe" and "tube". The value `relation` holds the relation (here the "hypernym") from left ("pipe") to the right ("tube"). "pipe is a tube", thus left hand side entails right hand side. The relation string holds the relation

name as it is written in the resource (it is WordNet here). In this case, the confidence is not provided by the implementation, and it is filled in with `DEFAULT`. Additional information is provided in the example with info variable. It holds an object called "WordNetInfo", which holds information that is depending on the lexical resource.

On the right, another example shows that "Magritte" entails "painter". In this case the knowledge is captured from Wikipedia, and the resource dependent information is filled with a different information object. Here, the implementation also provides a confidence value.

All lexical knowledge of EXCITEMENT resource will be delivered as an instance of this object. Java source code of the `class LexicalRule` is provided in Section C.5, "`class LexicalResource` and related objects".

4.6.1.1. Variables and access methods.

- `private final String leftLemma`: lemma of the LHS (left hand side).
- `private final PartOfSpeech leftPos`: POS of the LHS.
- `private final String rightLemma`: lemma of the RHS (right hand side).
- `private final PartOfSpeech rightPos`: POS of the RHS.
- `private final I info`: An object of type I, which will hold additional information that can be vary among knowledge bases.
- `private final DecisionLabel relation`: This variable holds an enum value of `DecisionLabel`(Section 4.2.1.6, "enum `DecisionLabel` "). It is expected only the top enum values of the enum hierarchy (`Entailment` and `NonEntailment`) will be used in this variable.
- `private final String originalRelation`: If the resource uses some relations (like WordNet or VerbOcean), this string holds the relation name as string. Otherwise, null.
- `private final String resourceName`: name of the resource
- `private static final String DEFAULT_CONFIDENCE`: A class variable that holds a value indicating that no confidence is available.
- `private final double confidence`: The confidence score assigned to the rule, in the interval [0,1]. It is assumed that higher values mean higher confidence. If no meaningful confidence score is provided by the resource, this value will hold the value of DEFAULT_CONFIDENCE.

For each variable, a corresponding access method is provided (for example, `getLeftPos()` for `leftPos`). The object is immutable, and the values can be only set by the constructors.

4.6.1.2. Related Types

- `interface RuleInfo`: An interface reserved for additional information. Any additional information that is not covered in the `LexicalRule`, depending on the specific rule base, should be included by implementing this interface. .
- `abstract class PartOfSpeech`: This class represents a generalization for part of speech tags. By implementations, it can support different tag-sets. The platform will provide common canonical POS set that is corresponding to the POS type of adopted CAS types. It is expected that each knowledge base will express the POS information according to the canonical set of POS labels.

4.6.2. Interface `LexicalResource`

```
public interface LexicalResource<I extends RuleInfo>
```

A lexical resource is a collection of lexical rules of a certain type (like WordNet, or VerbOcean) which can be queried. The interface provides three query methods. Queries are specified by lemma and POS pairs,

and the results are returned as a list of `LexicalRule` objects. The Java source code of the interface is listed in the Section C.5, "`class LexicalResource` and related objects". `LexicalResource` is a subinterface of `Component` and adds the following methods:

- `List<LexicalRule<? extends I>> getRulesForLeft(String lemma, PartOfSpeech pos)`: Return a list of lexical rules whose left side (the head of the lexical relation) matches the given lemma and POS. An empty list means that no rules were matched. If the user gives `null` POS, the interface will retrieve rules for all possible POSes.

- `List<LexicalRule<? extends I>> getRulesForLeft(String lemma, PartOfSpeech pos, DecisionLabel relation)`: an overloaded method for `getRulesForLeft`. In addition to the previous method, this method also matches the `relation` field of `LexicalRule` with the argument.

- `List<LexicalRule<? extends I>> getRulesForRight(String lemma, PartOfSpeech pos)`: Return a list of lexical rules whose right side (the target of the lexical relation) matches the given lemma and POS. An empty list means that no rules were matched.

- `List<LexicalRule<? extends I>> getRulesForRight(String lemma, PartOfSpeech pos, DecisionLabel relation)`: an overloaded method for `getRulesForRight`. In addition to the previous method, this method also matches the `relation` field of `LexicalRule` with the argument.

- `List<LexicalRule<? extends I>> getRules(String leftLemma, PartOfSpeech leftPos, String rightLemma, PartOfSpeech rightPos)`: This method returns a list of lexical rules whose left and right sides match the two given pairs of lemma and POS.

- `List<LexicalRule<? extends I>> getRules(String leftLemma, PartOfSpeech leftPos, String rightLemma, PartOfSpeech rightPos, DecisionLabel relation)`: an overloaded method for `getRules`. In addition to the previous method, this method also matches the `relation` field of `LexicalRule` with the argument.

The implementation must return an empty list (not `null`), if no applicable rules are found.

4.7. Interface of syntactic knowledge components

This section deals with syntactic knowledge (for example, tree rewriting rules). The basic unit of knowledge is a `SyntacticRule`. A collection of such rules is described by the interface `SyntacticResource`.

4.7.1. Type `SyntacticRule`

```
public class SyntacticRule
```

The class represents an instance of syntactic knowledge.

4.7.1.1. Variables and access methods

- `BasicNode leftHandSide`: LHS of this rule. A parse tree represented with `BasicNode`. The value is set by the constructor, and can be read by `getLeftHandSide()`.

- `BasicNode rightHandSide`: RHS of this rule. A parse tree represented with `BasicNode`

- `BidirectionalMap<BasicNode,BasicNode> mapNodes`: a bidirectional mapping between two parse tree nodes of type `BasicNode`.

For each variable, a corresponding access method should be provided (for example, getMap(), etc). If such an implementation is possible, the object should be immutable, and the values can be only set by the constructors.

4.7.1.2. Related Objects

- `BidirectionalMap<K,V>`: A map similar to normal hash map, but it also has back-links (thus, bidirectional mapping). The specification do not specify about the implementation. It can be any working bidirectional map, like that of apache commons, google collection, or BIUTEE infrastructure.

- `BasicNode`: This class represents a generic node in a dependency parse tree. The specification adopted BIUTEE's syntactic representation as common EXCITEMENT syntactic representation. It is described in the next section.

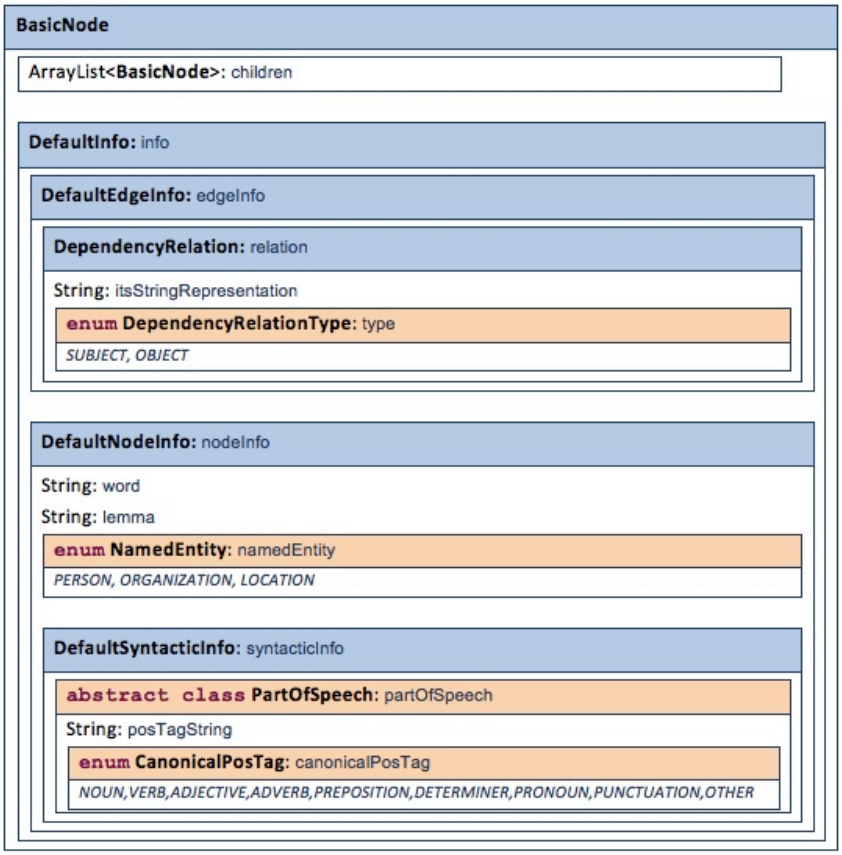
Note that the class `SyntacticRule` is described as a simple class, but its implementation can actually be a generic class that is parameterized by `BasicNode` and its related objects that is described in the next section.

4.7.1.3. Basic Node: Common representation for dependency parse tree

Common syntactic representation is required to share common representation for syntactic knowledge. The class `BasicNode` provides the common syntactic representation of EXCITEMENT. It represents a dependency parse tree node, linked to other parse tree nodes. By acquiring the top node, it can also represent a parse tree, or a partial parse tree.

Figure 8, “Contents of a basic node” shows an graphical overview of the basic node.

Figure 8. Contents of a basic node



A node basically holds two set of member variables. One is about tree itself (parent-children relation), and the other is information on edges and nodes.

- `children`: this member variable holds an array of child-nodes. They are nodes of the same type. Note that the node does not explicitly keep its parent.

- `info`: this member variable holds various information of its edge (incoming edge from the parent), and the node content.

The member `info` consists of two parts: `edgeInfo` holds information on edge, and `nodeInfo` has node content. `edgeInfo` holds the following information.

- `relation`: this is an instance of data object that represent the dependency relation. The instance holds two variable. One is `itsStringRepresentation` that holds the relation output of the dependency parser. And the other is `type` which will hold an enum value that represents simplified dependency relationship. It will hold SUBJECT, OBJECT, or null.

The `nodeInfo` holds the following information:

- `word`: this string holds the word (token) itself.
- `lemma`: the lemma of this token
- `namedEntity`: a simple enum that represents named entity. It will hold PERSON, ORGANIZATION, LOCATION, or null.
- `syntacticInfo`: this variable holds information about POS tag of this token. It holds two variables: the string output of the POS tagger (`posTagString`) and the normalized POS tag (`cannonicalPosTag`) that is equivalent to POS types of UIMA CAS.

Note that the various info objects of the `BasicNode` are supposed to be implemented with generics (For example, every name that starts with "Default" in the figure). One such implementation adopted from BIUTEE, is listed in the Section C.6.2, “class `BasicNode` and related classes”.

4.7.2. interface `SyntacticResource`

A syntactic resource is a collection of syntactic rules. For syntactic rule collections, it is not practical to provide simple access interfaces like `getLeftForPOS` for lexical knowledge. Due to the exponential number of subtrees in any text and hypothesis, naive querying based on the input is infeasible. Instead, `findMatches()` method is defined to outline common behavior of platform syntactic rules.

4.7.2.1. Methods

`SyntacticResource` is a subinterface of `Component` and adds the following method:

- `List<RuleMatch> findMatches(BasicNode currentTree)` The interface takes in a parse tree (which is represented in common parse tree nodes). The rule base must return all possible rules that can be applied into the tree, as a list of `RuleMatch` object. Note that the returned match holds information of not only rules (instance of `SyntacticRule` but also the location of the place where the rule should be applied to.

The implementation must return an empty list (not `null`), if no applicable rules are found.

4.7.2.2. Related Objects

- `RuleMatch`: this simple object represents a match of a syntactic rule. It has two variables. Member variable `rule` is a `SyntacticRule`, which holds the rewriting rule. The other variable `matchOn` is a reference to `BasicNode`, and this reference points a parse tree node where the rule can be applied upon.

4.8. Initialization and metadata check

4.8.1. Recommended policy on metadata check

Recall that a central goal of the EXCITEMENT platform is to provide a testing ground for various pluggable components that decide textual entailment. This is done by providing a set of common interfaces, and common data representations. Another requirement is to provide multilingual support.

From this perspective, compatibility check becomes a necessity, since components do not know in what context they are being called. Each component should provide a minimum level of compatibility check, on its configurations as well as on its inputs. This subsection describes recommended policy of metadata checking for the EDA, the core components, and between the EDA and the core components.

Compatibility checks can be classified into two groups. One is **checks at startup time** (configuration setting and initialization time). The other is **checks at data processing time**. The general guideline is that EDAs should do both types of checks, and components should at least do the startup time check.

An EDA must perform a compatibility check at its initialization time. This is a two-step process. First, it should check the provided configuration and check that the provided configuration is compatible with the EDA. For example, the language to be processed is supported, and parameters of the EDA is valid, etc. Second, the EDA must initialize its sub-components, by calling `initialize()` or an equivalent method of the used components. Each sub-component, in turn, will do their own checks, using the information from the common configuration file.

Thus, two types of metadata check failure can happen at the EDA initialization time. First, the EDA itself can report an exception, like inability to process a certain language, inability to provide a mode (like multi-text entailment), missing parameters, etc. The sub-components of the EDA can report exceptions as well. In this case, the EDA must hand through the exception to the user code. Thus, the successful completion of the EDA initialization must be interpretable to mean that all sub components are successfully initialized and are ready to process input.

An EDA must also perform a compatibility check at the data processing time, when its `process` method is called. The input to EDA `process()` is a CAS (as JCas object). EDA must check the validity of the input. At least two things should be checked, namely its language (meta data at document annotation type of CAS), and the existence of the analysis layers required by the components (like POS tagging, or dependency parsing, etc).

Components do not need to do additional checks at the process time. Once a component is properly configured and initialized with a configuration, it is the EDA's responsibility to use the component properly within its capabilities. However, a component may choose to provide meta data checking at processing time.

4.8.2. Interface `Reconfigurable`

Some components may need the capability of reconfiguration. For example, a user of a distance calculation component wants to get 10 feature values, by using 10 different "weighting scheme", from one distance calculation component. One way of doing this would be generating ten instances of the same component. However, this might not be desirable in certain cases; if the component does not permit more than a single instance, or if the initialization of the component takes a lot of time.

For such a case, we need a common method to change only a partial part of the component configuration. The interface `Reconfigurable` provides a common way of achieving this. The term "reconfiguration" here means changing a part of configurable parameters of a component, without re-initializing all of the resource, or the configurations that is already set for the components.

Any entailment core component that supports reconfiguration should implement this interface.

4.8.2.1. Methods of Interface Reconfigurable

```
public void reconfigure(CommonConfig config)
```

The interface provides a single method, `reconfigure()`. It shares the same single argument with `initialization()`, an instance of `CommonConfig`. However, the contract is different. Instead of copying and initializing all of the passed configuration value, the implementation must check the passed configuration and only reconfigure the changed values.

Note that it is the component's responsibility to check the consistency of the passed configuration value. If the passed value is inconsistent (for example, configuration values that cannot be reconfigured, are

different from the one given from `initialize()`), the implementation should raise an exception. Also, any component that supports this interface must clearly state in its documentation which configuration parameters are reconfigurable, and which are not.

4.8.3. Component name and instance name

All entailment core components and EDAs provide `initialize()` methods. Components may provide the `reconfigure()` method as well. The methods uses a `CommonConfig` instance as the argument. This central configuration scheme assumes that each component knows its own name, and it can retrieve corresponding configuration data from common configuration by querying the configuration with its name.

Component name is a `String` value that is designed to be read and written by users. This name must be able to uniquely identify a component or EDA within the EXCITEMENT framework. This name will be used in the configuration files to denote a section that describes the configuration for the component. All entailment core components and EDAs must have this component name string. Conversely, for each component that is initialized there must be a corresponding subsection in the configuration file. The components share a common way of access the names (`String getComponentName`). To see the use case of names within configuration interface, see Section 5.1, “Common Configuration ”.

Instance name is a `String` value that is needed for components that permits *multiple instances with different configurations*. This name will be used in the configuration files to denote a subsection that describes the configuration for the instance. Conversely, for each instance that is initialized there must be a corresponding subsection in the configuration file. The components share a common way of access the names (`String getInstanceName`). To see the use case of names within configuration interface, see Section 5.1, “Common Configuration ”.

The specification do not describe how the name are stored internally in the component implementations. It is expected that the platform implementation efforts will come up with a common best practice to keep the names in component implementations.

4.8.4. Initialization Helper

The main access point to the entailment core is the EDA interfaces. Users of the platform will process the data with LAP, and will use EDA interfaces to get the entailment results. To use an EDA, an instance of the EDA object must be first instantiated, and then it must be initialized with a proper configuration. Then, the EDA will initialize all needed components accordingly (see also Section Section 5.1.5, “Component selection”.

Thus, to start an EDA's initialization sequence, one must first instantiate an instance, and then feed it a configuration object. This also assumes that the configuration file is already read, and the main EDA that is defined in the configuration is already determined. `InitializeHelper` is an auxiliary layer that takes care of this instantiation and initialization. Without the helper, all of those process would have to be done by user code, which would require different initialization sequences for different EDAs.

`InitializeHelper` solves this problem by providing a common initialization capability. The helper code accepts a configuration file name, and then it reads the configuration file to determine the main EDA class. Then, the helper instantiate and initialize the EDA, and returns a ready-to-use EDA instance. [SP: how does this help know which components to initialize?? We don't know this -- unless we use one option in the configuration to specify what components to intialize.]

4.8.4.1. Methods of Initialize Helper

```
public EDABasic startEngine(File f)

public EDAMultiT startEngine(File f)

public EDAMultiH startEngine(File f)

public EDAMultiTH startEngine(File f)
```

`InitializeHelper` has a set of public methods. They all start the EDA initialization sequence with a File object (that has an opened XML configuration file). If the initialization was successful, it will return an EDA instance, as one of the EDA interfaces.

With the initialize helper, user code can simply open a valid configuration file, and call the initialize helper to get a prepared EDA. Once the `startEngine` is done, the user code can call `process()` of the EDA instance. Note that the user code must know what type of EDA interface it will use (i.e.. `EDABasic` or `EDAMultiT`) before hand, since they require different `process()` methods.

5. Common Data Formats

5.1. Common Configuration

5.1.1. Requirements of Entailment Core Common Configuration

In order to make an easy combination of EDAs and components possible, the configurations of all entailment core components should be stored and accessed in a uniform manner. A framework for the EXCITEMENT common configuration must meet the following requirements:

- The common configuration holds "all configurations for all things" in the Entailment Core.
- Its fundamental functionality is an attribute-value list (aka hash table, aka dictionary) with typed values (Strings, Numbers).
- This configuration exist as an in-memory object, as well as in human-readable and -modifiable file(s).
- Each individual module (EDA and components) can access the provided in-memory configuration.
- Independence 1: Each component has a dedicated region in the common configuration. A region means "a part of the configuration object that is devoted to a component". Each region is unique and unambiguous, and they are one-to-one relationship with core components. Each region has its own namespace. It should not be affected by name-value pairs of other regions.
- Independence 2: Components can change their own configuration over the course of their runtime without having to update the central configuration object. Thus, other components should avoid using the configuration of a component to make assumptions about its runtime configuration. This caveat explicitly does not apply to the configuration of the LAP, which we assume to remain unchanged.
- Support for multiple instances: Some components need to be deployed in multiple instances, with different configurations. The common configuration should be able to support such cases.
- Support for global information: There is also a special "global" region. This region does not correspond to a component, but defines *platform-wide* options, like current language, the selected top-level EDA, or other global options. Note: this section does *not* hold information about selected components; this information has its place within the EDA section/subsection. See Section 5.1.5, “Component selection”.
- Support for import, variables, and system environments: It is assumed that the XML configuration file will support file import (ability to store configurations in a set of files), variable support (within the configuration XML file, not within common configuration memory object), and inclusion of system environment variables.

5.1.2. Overview of the common configuration

This specification assumes that the platform shares a single implementation of configuration access. There are several stable implementations which satisfies the technical assumptions of configuration library that is described above. Apache common config [Apache Configuration], or JConfig [JConf], supports them. This specification does not assume a specific underlying library; we assume that the minimal functionality that we describe can be provided by several standard implementations.

We start with a set of design decisions that follow from the requirements above.

- Components need to be able to recognize relevant sections in the configuration. Thus, each component class must have a *component name*. This name should be easy for users to read and

write in a configuration file. At the same time, individual instances of components require an *instance name* to distinguish among multiple instances. We recommend that component and instance names are human-readable rather than UUID-like.

The common configuration object is able to return the applicable region of the configuration object for a component and for a component instance. The global section, with its special status, should be named `PlatformConfigurationRegion`.

- Hierarchical data storage: The common configuration data structure is a recursive attribute-value list. That is, attributes cannot only have types atoms as values, but also attribute-value lists themselves. We foresee at least two levels: the level of "sections" and the level of "subsections". These levels correspond directly to components and instances: Sections contain information about components and are labeled with components IDs. Subsections contain information about instances and are labeled with instance IDs.

  Note that (unlike in the existing EDITS configuration), the plain existence of a section or a subsection does not automatically means initialization of the component corresponding to that configuration. For example, a configuration file might have configuration sections of three EDAs, and choose one EDA to run, by specifying that EDA in the framework section as main EDA. This also makes it permissible to specify different "possible" or "recommended" configurations of a component as subsections in the configuration, only one of which is finally initialized.

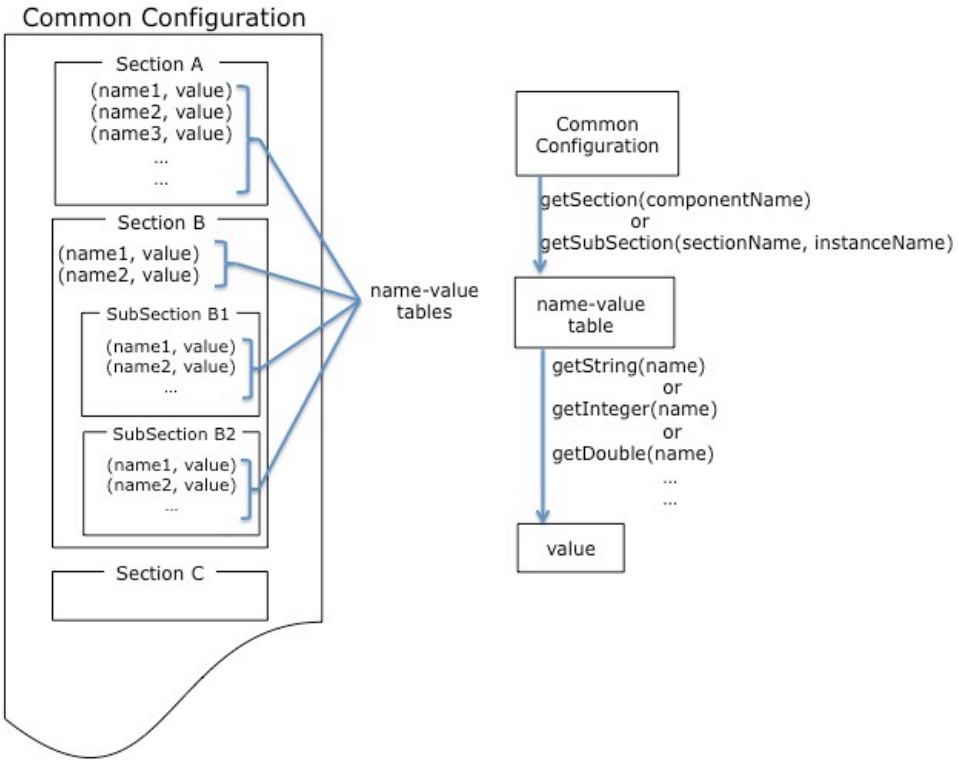
- Storage in XML format: The human-readable version of the common configuration will be stored and read as XML format. Users can edit the XML file as ordinary XML file. Since names are supposed to be unique, a name collision leads to an exception when the XML serialization of the configuration object is loaded.

- Typing: All objects in XML are strings. The configuration currently does not know about the value types for its keys. Rather, we achieve typing by providing different access functions to the configuration. It is currently the user's responsibility to ask the configuration for the value of a key as a specific type.

- The common configuration object specifies the behavior of components *at initialization and/or reconfiguration time*. The common configuration object is passed to each component as the argument of the `initialize()` and `reconfigure()` methods. Each component must read the configuration and initialize and/or reconfigure itself accordingly. [SP: we need to check that the changes in 4 match with this.]

Figure 9, “Overview of Common Configuration” shows a conceptual overview of common configuration object. On the left hand side, the common configuration object is shown as a big table with hierarchy. It has two depths, sections and subsections. Each section is corresponding to a "region" for a specific entailment component. Actual configurations values are represented as lists of "(Name,Value)" pairs. A set of name-value pairs are called as a name-value table in this section.

Figure 9. Overview of Common Configuration



On the right hand side of the figure, the access of configuration values are described on a conceptual level. A component can access an instance of the common configuration object with `getSection()` or `getSubSection()` methods for components and their instances, respectively. A successful call returns a name-value table. Each value associated with a name can be accessed by a get method.

Each section and subsection is represented as a name-value table. Once retrieved as a table, there is no differences between subsection and section. Note that one cannot access subsections of a section from a name-value table. A name-value table only holds the list of name-value pairs. The table is a typed key-value list. Thus, a value (which was written in XML file as a string) can be accessed with `getString(name)`, `getInteger(name)`, or `getDouble(name)`. Automatic conversion from XML string will occur, and the value will be returned as a requested type. (For the moment, the configuration itself

As an in-memory object, an instance of common configuration object is a single big block. However, in XML file level, they can actually come from more than single sources. For example, a section might exist in in a separate file and may have imported in the main XML file. This type of import support is already well implemented in existing configuration libraries.

As an in-memory object, all name-value pairs are simply seen as a name (string) and an associated value. However, in XML file level, they can actually come from variables defined in the XML file (like, <property name="something">&variable</property>). Also, the configuration file should easily encode current system environment values in the XML configuration file (For example, it should permit expressions that uses $PATH variable, like <property name="mypath">{$PATH}/etc</property>). Many configuration management libraries already support such features. (Note. The above pseudo XML codes are given only as examples. The specification does not assume anything about how the name-value pairs are represented in XML format.)

For all entailment core components, the configuration is delivered as a whole (single big block contains all pairs) via `initialization()`. It is the responsibility of the initialization (thus of its implementers) to retrieve and use the relevant sections (by using getSection or getSubSection).

All instances of a component might share a single configuration. In this case, there will be no subsections. Other components may need multiple instances with different configurations. They can

utilize sub-sections to keep various configurations for different instances. Configurations that will be shared among all instances can be written in the section area, while instance-dependent configurations can stay in the sub-sections. There is, however, no automatic inheritance of section-level information at the subsection level. That is, component instances will typically have to query both the section level for component-wide configuration and the subsection level for their own instance configuration.

5.1.3. Interfaces related to Common Configuration

5.1.3.1. Class `CommonConfiguration`

The section describes the public interfaces that need to be exposed to component authors. (Note that for the moment, the specification does not provide any method that can iterate over sections. For example, one cannot query a common configuration object to get all section names. This was intentional in current approach.)

5.1.3.1.1. method `getSection()`

`public NameValueTable getSection(String component Name)` This method returns the name-value table that is associated with the `componentName`. If there is no such section, the method will raise an exception.

5.1.3.1.2. method `getSubsection()`

`NameValueTable getSubSection(String componentName, String instanceName)` This method returns the name-value table that is associated with the `componentName` and `instanceName`.

5.1.3.1.3. method `loadConfiguration()`

`void loadConfiguration(File)` This method loads a configuration from a XML File. If the target file is not valid (invalid XML or missing mandatory section), the call will raise an exception.

Note that this `loadConfiguration()` is not supposed to be called by core components. The `EngineStartup` component (section []) will get the configuration file path and load the file with this call. And then, `EngineStartup` will generate the selected EDA and and pass the configuration as the argument of `initialize()`.

5.1.3.1.4. method `saveConfiguration()`

`void saveConfiguration(File)` This method saves current configuration to a XML file. It will save whole values as a single XML file.

Note that `saveConfiguration()` method is provided mainly for user level or transduction layer level access. The methods are not expected to be called from a entailment core component.

5.1.3.2. class `NameValueTable`

All configuration parameters are stored as name-value pairs in a table of class `NameValueTable`. The table can be read by a set of access functions.

5.1.3.2.1. get methods

```
public String getString(String name)

public Integer getInteger(String name)

public Double getDouble(String name)
```

Configuration data stored in an XML file are written as text string. XML parsers can recognize XML primitive types like string, boolean, decimal, double, etc. In this specification, we will deal only a few primitive types, like string, integer and double.

All get methods have a single string argument, the name part of a name-value pair. A get method returns the corresponding value from the name-value pair. Each get method will try to convert the XML value into requested type. If the conversion fails, the get method will raise a conversion exception (one of ConfigurationException). [SP: Ideally I would say that each EDA should implement a static method "getOptions" which return all options with their types (currently String, Integer, or Double). Then the access functions can check the type internally (!), AND we can also find inconsistencies in the configuration file: missing (=unspecified!) options, and spurious options that aren't used. What do you think?]

5.1.3.2.2. Set methods

`public setString(String name, String value) public setInteger(String name, Integer value) public setDouble(String name, Double value)`

Set methods are provided for editing existing values or writing new values. The values added/modified by set methods will only affect the XML file by `saveConfiguration()`.

Note that set methods are provided mainly for user level or transduction layer level access. The methods are not expected to be called from a entailment core component.

5.1.4. Extending the common configuration features

This section defines is a minimalistic approach to configuration management. We leave possible extensions (like default values or possible ranges) to future versions.

5.1.5. Component selection

The responsibility for component selection -- i.e., the decision which components to activate for a given run of the engine -- could be assigned to different layers: to the user level code, to the initialization helper, or to the top-level EDA. We make it a responsibility of the EDA, in order to allow EDAs to "hide" some complexity to its users.

Consequently, the selection which components to use forms part of the configuration of the EDA and is specified in the EDA section and/or subsection. We expect EDAs to specify how the set of components to be activated is specified as a configuration option. (Note that currently the configuration does not lists). It is the responsibility of the EDA, not of the user level code, to initialize these components.

5.2. Input file format

5.2.1. Role of input data

This section describes the raw input data. By "raw input data" we mean text-hypothesis pairs without linguistic analysis layers. The RTE dataset formats (RTE1-5) are the most widely used file format for this purpose.

The internal EXCITEMENT entailment problem representation is the CAS representation described in Section 3. The raw input data formats only serve as as basis from which the CAS representations is created. Each linguistic analysis pipeline (LAP) *must* provide readers for the RTE dataset formats that produce CAS objects from corresponding files. All RTE formats to be supported are listed in Appendix D, *Supported Raw Input Formats*. [SP: or should be make just a subset obligatory?]

NB. RTE1-5 only support the encoding of "classical" (single H, single T) entailment problems. That is, LAPs are only required to deal with classical entailment problems. Support for multi H-T or T-multi H problems is optional. Each LAP should specify clearly in its documentation which additional problem types it supports and what file format it expects or these problems (such as RTE-6 or later).

The following section describes the RTE data formats, and adds a simple modification.

5.2.2. RTE challenge data formats and the supported data format.

The RTE challenge data formats are the most well known and widely used data format among textual entailment community. RTE-1 to RTE-5 formats are focused on single T-H pairs.

One problem of RTE format is the lack of language marking. Language designation is important in the EXCITEMENT platform. Thus we adopted a tiny modification to RTE format. The following is the DTD of RTE-5 main task data representation.

```
<!ELEMENT entailment-corpus (pair+)>
<!ELEMENT pair (t,h)>
<!ATTLIST pair
         id CDATA #REQUIRED
         entailment (ENTAILMENT|CONTRADICTION|UNKNOWN) #REQUIRED
         task (IR|IE|QA) #REQUIRED >
<!ELEMENT t (#PCDATA)>
<!ELEMENT h (#PCDATA)>
```

It defines an XML format that holds top element `entailment-corpus`. The element can have one or more `pair` elements, which have t and h as string data. A `pair` must have a set of attributes. Here they are `id` (identifier of the pair, as string), `entailment` (the entailment decision), and `task` (as one of information retrieval, information extraction, or question answering).

We modify the DTD as follows:

```
<!ELEMENT entailment-corpus (pair+)>
<!ATTLIST entailment-corpus
          lang CDATA #IMPLIED>
<!ELEMENT pair (t,h)>
<!ATTLIST pair
          id CDATA #REQUIRED
          entailment (ENTAILMENT|CONTRADICTION|UNKNOWN) #REQUIRED
          task (IR|IE|QA) #REQUIRED >
<!ELEMENT t (#PCDATA)>
<!ELEMENT h (#PCDATA)>
```

Now the top element `entailment-corpus` has an optional attribute `lang`. This attribute holds the language identification string according to ISO 639-1 [ISO 639-1]. For example, "EN" for English, "DE" for German, and "IT" for Italian. Since it is an optional (designated as #IMPLIED) attribute, existing RTE-5 XML data will be processed as a valid XML for this DTD. In that case, no language checking will take place.

Support for RTE-5 format is mandatory. EDA implementers must support processing of RTE-5 format. Other RTE formats are optional. EDAs (and their LAPs) may support them.

At the current stage (non-decomposed pipelines), each LAP will have to implement its own readers. (It is permissible to have one reader that can deal with all RTE1-5 formats, given the large similarities between the formats.) In the future, we foresee that RTE1-5 readers will be provided as a "collection reader" component of UIMA component, and it is expected that such components will be implemented only once and shared among all EDAs and EDA implementers.

6. Further Recommended Platform Policies

6.1. Coding Standard

This section describes the coding standard for EXCITEMENT platform. The coding conventions are adopted from the coding guideline of the BIU NLP lab. (Only the code annotation part of the guideline is missing, since it needs an additional common code base. -- This, and related issues on the coding standard will be discussed in the implementation process.)

6.1.1. Documentation

- Every class / interface / enum should include a main comment in its beginning, describing:
    1. the purpose of the class
    2. its contents
    3. its usage
    4. and how it is related to other classes.
- The main comment should include also the author name and the date when the class was created first.
- Each method must be paired with a comment describing its purpose, usage, parameters, return value. In particular, the comment must make explicit as any policies on its use beyond its type signature.
- The class and functions (methods) comments specified above should be in Java-Doc format.
    - Tip: in Eclipse use Alt+Shift+J to generate a javadoc skeleton.
- Comments should be written in the code itself, describing the flow, i.e. describing what the code does. It is required when the code is not clear (i.e. self explaining), and recommended for any long code (long code = code with more that 5 lines).
- The best practice is writing in such a way that a programmer that will read your code will be able to understand it, without additional explanations.
- Packages and non-trivial fields should also be documented.

6.1.2. Error Handling

- Make sure you're familiar with the concept of Checked Exceptions in Java [CheckedEx].
- We will use a single way to handle errors: throwing exceptions.
- Code should not write to `System.err`.
- Code should not call `System.exit()`.
- Writing to `System.err` is allowed only in the module that handles the very beginning and very end of the flow (i.e. the class that contains the `main()` method, and may be one or two other classes that are called by it).
- Calling `System.exit()` is allowed only for GUI applications, and only in the module that handles the GUI events.

- `RuntimeException` and its subclasses should never be thrown explicitly. It is also recommended to wrap implicit potential throws of subclasses of `RuntimeException` by `try...catch` that throws a subclass of Exception that is not a runtime exception.

- When throwing an exception, include a string in the exception that describes the problem, and how it can be fixed.

- "Stopping" an exception is usually a very bad idea. "Stopping" means is catching it somewhere and not throwing it (or another exception) again. The problem is that the user will not be aware of underlying the problem that caused the first exception to be thrown.

- Handling exceptions that are thrown from an inner components can be done in two ways:

    1. Catch them, and throw a new exception that wraps the original ones.

    2. Let them be thrown up in the call-stack.

6.1.3. Naming Conventions

- Package names should be in lower case letters only. Even a multi-word name should not include any upper-case letter.

- Class names must start with an upper case letter.

- Function names and variable names must start with lower case letter.

- Constant names must include upper-case letters only, No lower case letters are allowed in constant names. The underscore (_) character can be used to separate words in constant names.

- Use meaningful names for everything.

6.1.4. Writing Good Code

- Local variables should be declared in the inner-most possible block. Don't declare local variables in advance, but ad-hoc. This convention helps eliminating some hard-to-observe bugs.

    1. Nevertheless, for the sake of saving many calls to a costly constructor, it may be wise to declared a local variable out of its minimal scope

- Write short code.

    1. Classes should be short. Try writing classes that are no longer than 300 lines. A long class is an evidence to poor design. A long class is usually a class that had to be created as several classes, in a hierarchical way.

    2. Functions should be short - no more than 25 lines. A long function is hard to understand, and is an evidence that one function does too many things, that had to be partitioned into several functions (most of them non-public).

- Do not use nested classes. Nested classes are required only in rare cases, where the nested class needs access to its parent's private members , should not be known outside, and is logically part of the parent, but needs also its own private context. Those cases are rare.

- Do not use nested static classes. You can use them only sometimes for declaring specific exceptions, or in some rare cases. In general, using nested classes, either static or non-static, makes a hard-to-understand and hard-to-change code.

- Make sure your code has no compilation warnings. Compilation warnings are a good tool for avoiding bugs. A code that contains warning makes them unusable.

- It is strongly recommended that every class will contain all of its "public" constructors / methods and fields together. Putting all of the public stuff at the beginning of the class, with a clear comment separator between public and non-public part, makes the class easier to understand and use.

    1. It's convenient to change eclipse's settings to place new generated private methods at the bottom of the class.

- Use constants. Numbers and strings should not be hard coded in the code itself, but as constants. Put all the constants together at the beginning of the class, and make them "final".

- Never use early-access code or any code that may become incompatible in future environment.

- For *abstract data types*, that is, classes that represent some non-atomic information as a single object, make sure that the following conditions are met:

    1. Abstract data types should typically implement their own "equals" and "hashCode" methods. Make sure that they are implemented if necessary.

    2. Make sure you do not implement those methods when they should not be implemented.

    3. Make sure you implement them correctly. Eclipse has a default way to implement those methods (source--> insert --> hashcode feature). Use it. Do not use another implementation unless you know what you are doing.

Write modular modules.

1. For each module, think that it can be used in another context than you originally intend to use it.

2. For each module, think that it can be replaced by another module with the same interface.

3. Write clear and simple interfaces for any module. A simple interface consists of a relatively small set of functions, that take very few parameters. UNIX system calls are a very good example of a very small set of function (about 100 functions), each takes very few parameters (1 or 2. Only one function takes 3 parameters). That set supplies all of the required functionality that an OS should supply.

4. A module should not be aware of any module that is not logically connected to it. In other words: If a module X is not necessary for the definition of another module Y, than Y should not refer to X in any way.

6.2. List of Exceptions

Exceptions are part of the method signature, as much as inputs and return types. However, it is not easy to specify enough details on exception hierarchy of EXCITEMENT platform in this early stage. Thus, in this specification, we will only try to outline them. The outlined objects will form the top level objects in the EXCITEMENT exception hierarchy. More detailed use cases and additional exceptions will be added along the way during the development of EXCITEMENT platform (WP4).

The following exceptions are assumed in the top of the exception hierarchy of entailment core. They are objects directly inherits java.lang.Exception (or EXCITEMENT platform-wise base Exception object).

- ConfigurationException: Interfaces of common configuration can throw exceptions of various kinds. Exceptions originated from common configuration code, and that can be checked, should use or inherit this exception.

- `EDAException`: Interfaces of EDA, and working codes of EDA can generate this type of exceptions. All checked exceptions thrown from EDA code should use or inherit this exception.

- `ComponentException`: This exception present an exception caused within an entailment core component. It is the base-type that will be inherited by core component exceptions like

`KnowledgeComponentException`, `DistanceComponentException`, or that of future core components.

- `KnowledgeComponentException`: Working codes of knowledge components (lexical resource and syntactic resource) can raise this exception.
- `DistanceComponentException`: Working codes of distance calculation components can throw this exception.

The above exceptions are only covering the core components. An LAP component uses different set of Exceptions. LAP components use UIMA exceptions, and follows UIMA component exception policies. The UIMA AE uses one of the following exceptions; `ResourceConfigurationException`, `ResourceInitializationException`, and `AnalysisEngineProcessException`. And the message encoded with the exceptions must use a specific message digest format. For more information on UIMA AE exceptions, see [UIMA-exceptions].

6.3. Conventions for Additional Names

The EXCITEMENT platform will include many names, not only as Java objects but also as UIMA type names, and also configuration names. This section tries to summarize some common naming conventions.

- Longer names are preferred to shorter names: For example, use `customerAccountDataPath`, not `custAccDTPath`.
- Do not use dash (-) or underscore (_) unless you have a good reason: For example, `supportedLanguage` (feature name), or `supported.language` (configuration name) are preferred over "supported_language" or "supported-language". One exception might be naming of constants (like `PI_VALUE`), where uppercases are used, and the usage of a dot (.) is not possible.
- Class names, or class like things (like UIMA type) should begin with an uppercase letter: If they are a compound word, use the CamelCase. For example, "EntailmentType", or "CalculatingSemanticDistance".
- Member names, or member like names should begin with a lowercase letter: If they are a compound word, they should use lower camel case (like, "anotherFeatureName", "stem", "nextNode").
- In all cases, the names should clearly represent what the value of the name represents.

In summary, all naming should be consistent with standard naming convention. For UIMA types, treat a type just like a Java objects. Since a type is equivalent to a class. UIMA feature structures are equivalent to class members, and all features names should start with lowercases, etc. Also, treat name-value pairs of common configuration as member items within a category. Thus, all property names of name-value pairs should start with a lowercase.

7. References

Bibliography

[Apache Configuration] Apache Commons Configuration homepage. *http://commons.apache.org/configuration/ [http://commons.apache.org/configuration/]*

[CheckedEx] Java Checked Exceptions (Java Programming Wikibooks). *http://en.wikibooks.org/wiki/Java_Programming/Checked_Exceptions*

[CLEARTK] CLEARTK homepage. *http://code.google.com/p/cleartk/*

[DKPRO] DKPRO homepage. *http://code.google.com/p/dkpro-core-asl/*

[DocBook] OASIS Committee Draft 4.4 DocBook XML, 17 January 2005. *http://www.docbook.org/specs/cd-docbook-docbook-4.4.html*

[ISO 639-1] ISO 639-1 Codes for the representation of names of languages.*http://en.wikipedia.org/wiki/ISO_639-1*

[PredicateTruth] Predicate Truth Annotation. *A Manual Syntactic Rulebase for a Textual Entailment Recognition System [http://dl.dropbox.com/u/6327182/seminar%20work.doc]*

[JConf] JuliusLib JConf Configuration homepage. *http://www.sp.nitech.ac.jp/~ri/julius-dev/doxygen/julius/4.0/en/group__jfunc.html*

[RFC 2119] S. Bradner, Key words for use in RFCs to Indicate Requirement Levels, IETF (Internet Engineering Task Force) RFC 2119, March 1997. *http://www.ietf.org/rfc/rfc2119.txt*

[UIMA] Apache UIMA Homepage. *http://uima.apache.org*

[UIMA-CAS] Apache UIMA reference, section on CAS. *http://uima.apache.org/d/uimaj-2.4.0/references.html#ugr.ref.cas [http://uima.apache.org/d/uimaj-2.4.0/references.html#ugr.ref.cas]*

[UIMA-doc] Apache UIMA Documentation. *http://uima.apache.org/documentation.html*

[UIMA-exceptions] Apache UIMA Tutorial and Developers Guide, section on Exceptions. *http://uima.apache.org/d/uimaj-2.4.0/tutorials_and_users_guides.html#ugr.tug.aae.throwing_exceptions_from_annotators [http://uima.apache.org/d/uimaj-2.4.0/tutorials_and_users_guides.html#ugr.tug.aae.throwing_exceptions_from_annotators]*

[UIMA-ser] Apache UIMA reference, section on XMI CAS serialization. *http://uima.apache.org/d/uimaj-2.4.0/references.html#ugr.ref.xmi [http://uima.apache.org/d/uimaj-2.4.0/references.html#ugr.ref.xmi]*

[xml-assoc] James Clark. Associating Style Sheets with XML documents Version 1.0, W3C Recommendation 29 June 1999. *http://www.w3.org/1999/06/REC-xml-stylesheet-19990629*

Appendix A. Type Definition: types for general linguistic analysis

{For the moment, the type tables are not exhaust.}

{At the 1.0 spec release, exhaust type lists will be released with the XML definition file that can be directly used in UIMA and related tools}

A.1. Segmentation types

Segmentations types are types that denotes various textual units, lemma, heading, paragraphs, sentence, etc.

- `de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Sentence`
  - Description: An instance of this type annotates a sentence.
  - Supertype: `uima.tcas.Annotation`
  - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Paragraph`
  - Description: An instance of this type annotates a paragraph.
  - Supertype: `uima.tcas.Annotataion`
  - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Stem`
  - Description: An instance of this type annotates a stem. It has a feature that holds the stem as string.
  - Supertype: `uima.tcas.Annotataion`
  - Features
    - `value` (`uima.cas.String`): stem value.
- `de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token`
  - Description: An instance of this type annotates a token. It has features that points lemma, stem, and POS of the token.
  - Supertype: `uima.tcas.Annotataion`
  - Features
    - `lemma` (`de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Lemma`)
    - `stem` (`de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Stem`)
    - `pos` (`de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS`)

A.2. POS types

Annotations that are used to mark part-of-speech are defined by these types. They are aligned with some simple hierarchy.

- de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS
  - Description: This is the top type of POS annotation hierarchy. It has one feature, PosValue, which denotes the raw output of the POS tagger.
  - Supertype: uima.tcas.Annotataion
  - Features
    - PosValue (uima.cas.String)
- de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PP
  - Description: This type annotates PP (pre-positions).
  - Supertype: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
  - Description: This type annotates pronouns.
  - Supertype: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PUNC
  - Description: This type annotates punctuations.
  - Supertype: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
  - Description: This type annotates verbs.
  - Supertype: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADJ
  - Description: This type annotates adjectives.
  - Supertype: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADV
  - Description: This type annotates adverbs.
  - Supertype: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ART
  - Description: This type annotates articles.
  - Supertype: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS

    - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.CARD`
    - Description: This type annotates cardinal numbers.
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS`
    - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.CONJ`
    - Description: This type annotates conjunctions.
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS`
    - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.N`
    - Description: This type annotates nouns.
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS`
    - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN`
    - Description: This type annotates normal nouns. Note the supertype is noun, not POS.
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.N`
    - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NP`
    - Description: This type annotates proper nouns. Note the supertype is noun, not POS.
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.N`
    - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O`
    - Description: This type annotates interjections.
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.POS`
    - Features: (no features)

A.2.1. Extension of basic POS types

{In DKPro, there are other POS tags designed to work for specific texts. We will generally not adopt them in EXCITEMENTS. However, two examples are shown here as an example of a type extension. They are designed for tweet texts, and the modules that understand those types can take benefits from them. Moreover, previously existing generic codes that do not understand the extensions, still able to access them as generic O, and NP.}

- `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.tweet.EMO`
    - Description: This type annotates emoticons of tweet texts. Note the super type O.

- Supertype: `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O`
- Features: (no features)

- `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.tweet.NPV`
  - Description: This type annotates user ID of tweet texts. For example, "@tailblues", "@magritte", etc. Note the supertype NP.
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NP`
  - Features: (no features)

A.2.2. Mapping of tagger tagsets to the types

The following two lists show the mapping of English and German tags of Tree Tagger, to the DKPro POS tags. This type of mapping (what actually means, say, NN in German), should be provided by the pipeline implementers.

English Tags (Penn Treebank)

- CC: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.CONJ
- CD: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.CARD
- DT: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ART
- EX: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ART
- IN: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PP
- JJ: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADJ
- JJR: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADJ
- JJS: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADJ
- MD: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- NN: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN
- NNS: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN
- NP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NP
- NPS: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NP
- PDT: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ART
- PP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PP$: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- RB: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADV
- RBR: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADV
- RBS: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADV
- RP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PP
- SENT: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PUNC

- UH: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O
- VB: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VBD: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VBG: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VBN: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VBP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VBZ: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VH: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VHD: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VHG: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VHP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VHN: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VHZ: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VV: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VVD: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VVG: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VVN: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VVP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VVZ: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- WDT: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ART
- WP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- WP$: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- WRB: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADV
- *: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O

German Tags (STTS)

- ADJA: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADJ
- ADJD: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADJ
- ADV: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ADV
- APPR: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PP
- APPR: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PP
- APPO: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PP
- APZR: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PP

- ART: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.ART
- CARD: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.CARD
- FM: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O
- ITJ: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O
- KOUI: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.CONJ
- KOUS: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.CONJ
- KON: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.CONJ
- KOKOM: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.CONJ
- NN: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NN
- NE: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.NP
- PDS: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PDAT: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PIS: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PIAT: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PIDAT: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PPER: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PPOSS: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PPOSAT: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PRELS: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PRELAT: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PRF: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PWS: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PWAT: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PWAV: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PAV: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PR
- PTKZU: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O
- PTKNEG: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O
- PTKVZ: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- PTKANT: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O
- PTKA: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O
- TRUNC: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O
- VVIMP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VVINF: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V

- VVIZU: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VVPP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VAFIN: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VAIMP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VAINF: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VAPP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VMFIN: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VMINF: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- VMPP: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.V
- XY: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O
- $,: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PUNC
- $.: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PUNC
- $(: de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.PUNC
- * : de.tudarmstadt.ukp.dkpro.core.api.lexmorph.type.pos.O

A.3. Document Metadata

- `de.tudarmstadt.ukp.dkpro.core.api.metadata.type.DocumentMetaData`
  - Description: This type extends basic document annotation of UIMA CAS. It holds additional data as strings.
  - Supertype: `uima.tcas.DocumentAnnotation`
  - Features
    - `documentTitle` (`uima.cas.String`)
    - `documentId` (`uima.cas.String`)
    - `documentUri` (`uima.cas.String`)
    - `collectionId` (`uima.cas.String`)
    - `documentBaseUri` (`uima.cas.String`)
    - `isLastSegment` (`uima.cas.Boolean`)
- `de.tudarmstadt.ukp.dkpro.core.api.metadata.type.ProcessorMetaData`
  - Description: This metadata is not related to a SOFA, and extended from a `uima.cas.Top`. It adds metadata of the processor (annotator) to the CAS.
  - Supertype: `uima.cas.TOP`
  - Features
    - `instanceId` (`uima.cas.String`)
    - `name` (`uima.cas.String`)

- `version` (`uima.cas.String`)
- `annotatorImplementationName` (`uima.cas.String`)

A.4. NER types

These types are related to named entity recognization. It uses a top type, which represents a generic entity. Actual types (person, organiztion, etc) are provided by inherited types.

- `de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity`
  - Description: This is the top annotation for NER types. It is a subtype of UIMA annotation, and provides one feature `value`. The feature holds raw output of the NER recognizer.
  - Supertype: `uima.tcas.Annotation`
  - Features
    - `value` (`uima.cas.String`)
- `de.tudarmstadt.ukp.dkpro.core.api.ner.type.Nationality`
  - Description: This type represents an entity of Nationality.
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity`
  - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.ner.type.Norp`
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity`
  - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.ner.type.Ordinal`
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity`
  - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.ner.type.OrgDesc`
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity`
  - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.ner.type.Organization`
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity`
  - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.ner.type.PerDesc`

    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Percent
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Person
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Plant
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Product
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.ProductDesc
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Quantity
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Substance
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Time

    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.WorkOfArt
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Animal
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Cardinal
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.ContactInfo
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Date
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Disease
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Event
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Fac

    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.FacDesc
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Game
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Gpe
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.GpeDesc
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Language
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Law
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Location
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.NamedEntity
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.ner.type.Location

- Description:
- Supertype: de.tudarmstadt.ukp.dkpro.core.api.ner.type.Money
- Features: (no features)

A.5. Types for Dependency Parsing

These types are related to dependency parse results. There is a top Dependency type that represents dependency relations (actual relations are expressed as inherited types), and nodes are expressed as type Dependent. Each dependent points its Governor, and the dependency relation.

- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
  - Description: Subtypes of this type, represent the dependency relations. The type has three features. Two as tokens. One as string. Feature Governor points the governor word, and Dependent points the dependent word. String DependencyType holds the dependency type, as string outputted from the parser.
  - Supertype: uima.tcas.Annotation
  - Features
    - Governor (de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token)
    - Dependent (de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token)
    - DependencyType (uima.cas.String)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Governor
  - Description: This type represents a Governor.
  - Supertype: uima.tcas.Annotation
  - Features
    - Dependent (de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token)
    - Dependency (de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependent
  - Description: This type represents a Dependent.
  - Supertype: uima.tcas.Annotation
  - Features
    - Governor (de.tudarmstadt.ukp.dkpro.core.api.segmentation.type.Token)
    - Dependency (de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.NSUBJ
  - Description:
  - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency

    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.AMOD
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.DEP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.DET
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.POBJ
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.PREP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.CC
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.CONJ
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency

    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.NN
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.NUM
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.CCOMP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.NUMBER
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.CONJ_YET
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.COP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.ADVMOD
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency

    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.COMPLM
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.AUX0
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.NSUBJ
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.XCOMP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.POSS
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.DOBJ
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.AUXPASS
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency

    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.RCMOD
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.AGENT
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.APPOS
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.TMOD
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.PARTMOD
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.MEASURE
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.ACOMP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency

    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.PREDET
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.MARK
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.ADVCL
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.NEG
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.NSUBJPASS
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.PREPC
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.QUANTMOD
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency

    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.CSUBJPASS
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.CSUBJ
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.REL
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.PRT
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.PUNCT
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.PCOMP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.XCOMP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency

    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.EXPL
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.INFMOD
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.CSUBJPASS
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.ATTR
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.IOBJ
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.CONJP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.PRECONJ
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency

    - Features: (no features)

- `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.PURPCL`
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency`
    - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.PRED`
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency`
    - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.PARATAXIS`
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency`
    - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.ABBREV`
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency`
    - Features: (no features)
- `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.POSSESSIVE`
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.dependency.Dependency`
    - Features: (no features)

A.6. Types for Coreference Resolution

Coreference resolusion is annotated with `CoreferenceLink` type. An instance of the type annotates a span, and links its next (corefered) span. A number of links will form a chain, and the starting point of this chain is pointed by `CoreferenceChain` type.

- `de.tudarmstadt.ukp.dkpro.core.api.coref.type.CoreferenceLink`
    - Description: An instance of this type annotates a single co-reference link. It has two features. One is `next`, which points another coreference link, the other is `referenceType` that is a string that holds the string output of the coreference resolutioner.
    - Supertype: `uima.tcas.Annotation`

    - Features
        - `next` (`de.tudarmstadt.ukp.dkpro.core.api.coref.type.CoreferenceLink`)
        - `referenceType` (`uima.cas.String`)
- `de.tudarmstadt.ukp.dkpro.core.api.coref.type.CoreferenceLink`
    - Description: A set of `CoreferenceLink` will form a chain that is linked by `next` feature of the `Corefrencelink`. This type is

      coreference link. It has two features. One is `next`, which points another coreference link, the other is `referenceType` that is a string that holds the string output of the coreference resolutioner.
    - Supertype: `uima.tcas.AnnotationBase`
    - Features
        - `first` (`de.tudarmstadt.ukp.dkpro.core.api.coref.type.CoreferenceLink`)

A.7. Types for Semantic Role Labels

- de.tudarmstadt.ukp.dkpro.semanticrole.Predicate
    - Description: This type represents a predicate of semantic role labeling.
    - Supertype: `uima.tcas.annotation`
    - Features
        - `predicateName` (`uima.cas.String`): This feature represents the name of this predicate. It refers to the sense of the predicate in PropBank or FrameNet.
        - `arguments` (`uima.cas.FSArray`): This feature is an array of semanticrole.Argument. It holds the predicate's arguments.
- de.tudarmstadt.ukp.dkpro.semanticrole.Argument
    - Description: This type represents an argument of semantic role labeling.
    - Supertype: `uima.tcas.annotation`
    - Features
        - `argumentName` (`uima.cas.String`): This feature represents the name of this argument. It refers to the different types of arguments in string, like "A0", "A1", "AM-LOC", etc.
        - `predicates` (`uima.cas.FSArray`): This feature is an array of semanticrole.Predicate. This is a backward references to predicates that governs this argument.

Appendix B. Type Definition: types related to TE tasks

This appendix section formally defines the types introduced in Section 3.3.4, “Additional Types for Textual Entailment”. Note that the string `EXCITEMENT` used in this section will be replaced with actual name in the future.

B.1. Types related to entailment problems

- `EXCITEMENT.entailment.EntailmentMetadata`
  - Description: This type provides metadata for entailment problem.
  - Supertype: `uima.tcas.DocumentAnnotation`
  - Features
    - `task` (`uima.cas.String`): This string holds the task description which can be found in the RTE chellenge data.
    - `channel` (`uima.cas.String`): This feature can holds a string that shows the channel where this problem was orignated. For example, "customer e-mail", "online forum", or " customer transcription", etc.
    - `origin` (`uima.cas.String`): This metadata field can hold a string that shows the origin of this text and hypothesis. A company name, or a product name.
- `EXCITEMENT.entailment.Pair`
  - Description: This type represents a text-hypothesis pair.
  - Supertype: `uima.tcas.Annotation`
  - Features
    - `pairID` (`uima.cas.String`): ID of this pair. The main purpose of this value is to distinguish a certain pair among multiple pairs.
    - `text` (`EXCITEMENT.entailment.Text`): This feature points a `Text` instance, which represents the text part of this pair.
    - `hypothesis` (`EXCITEMENT.entailment.Hypothesis`): This feature points a `Hypothesis` instance, which represents the hypothesis part of this pair.
    - `goldAnswer` (`EXCITEMENT.entailment.Decision`): This feature records the gold standard answer for this pair. If the pair (and CAS) represents a training data, this value will be filled in with the gold standard answer. If it is a null value, the pair represents a entailment problem that is yet to be answered.
- `EXCITEMENT.entailment.Text`
  - Description: This type represents a text part of a T-H pair. This type annotates a text item within the TextView. It can occur multiple times (for multi-text problems)
  - Supertype: `uima.tcas.Annotation`
  - Features (no features)

- `EXCITEMENT.entailment.Hypothesis`
  - Description: This type represents a hypothesis part of a T-H pair. This type annotates a hypothesis item within the HypothesisView. It can occur multiple times (for multi-hypothesis problems)
  - Supertype: `uima.tcas.Annotation`
  - Features: (no features)
- `EXCITEMENT.entailment.Decision`
  - Description: This type represents the entailment decision. It is a string subtype. The type can only have one of "ENTAILMENT", "NONENTAILMENT", "PARAPHRASE", "CONTRADICTION", and "UNKNOWN" The type can be further expanded in the future.
  - Supertype: `uima.cas.String`
  - Features: (no features)

B.2. Types for Predicate Truth

- `EXCITEMENT.predicatetruth.PredicateTruth`
  - Description: This type represents a predicate truth value annotation.
  - Supertype: `uima.tcas.Annotation`
  - Features:
    - `value` (`PredicateTruthValue`): This represents the value of the annotation.
- `EXCITEMENT.predicatetruth.PredicateTruthValue`
  - Description: This type is provides labels for `PredicateTruth`. This type is a string subtype that only permites "PT+", "PT-", and "PT?".
  - Supertype: `uima.cas.String`
  - Features: (no features)
- `EXCITEMENT.predicatetruth.ClauseTruth`
  - Description: This type represents a clause truth value annotation.
  - Supertype: `uima.tcas.Annotation`
  - Features:
    - `value` (`ClauseTruthValue`): This represents the value of the annotation.
- `EXCITEMENT.predicatetruth.ClauseTruthValue`
  - Description: This type is provides labels for `ClauseTruth`. This type is a string subtype that only permites "CT+", "CT-", and "CT?".
  - Supertype: `uima.cas.String`
  - Features: (no features)
- `EXCITEMENT.predicatetruth.NegationAndUncertainty`
  - Description: This type represents a negation-and-uncertainty annotation.

- Supertype: `uima.tcas.Annotation`
- Features:
  - `value` (`NegationAndUncertaintyValue`): This represents the value of the annotation.

- `EXCITEMENT.predicatetruth.NegationAndUncertaintyValue`
  - Description: This type is provides labels for `NegationAndUncerntainty`. This type is a string subtype that only permites "NU+", "NU-", and "NU?".
  - Supertype: `uima.cas.String`
  - Features: (no features)
- `EXCITEMENT.predicatetruth.ImplicationSignature`
  - Description: This type represents an implication signature of a predicate.
  - Supertype: `uima.tcas.Annotation`
  - Features:
    - `value` (`ImplicationSignatureValue`): This represents the value of the annotation.
- `EXCITEMENT.predicatetruth.ImplicationSignatureValue`
  - Description: This type is provides labels for `ImplicationSignature`. This type is a string subtype that only permites one of the following strings: "+ / -", "+ / ?", "? / -", "- \ +", "- / ?", "? / +", "+ / +", "- / -", "? / ?".
  - Supertype: `uima.cas.String`
  - Features: (no features)

B.3. Types for Temporal/NER events

- `EXCITEMENT.temporal.DefaultTimeOfText`
  - Description: This type is anchored to a textual region (a paragraph, or a document), and holds the "default time" that has been determined for this passage and can be useful to interpret relative time expressions ("now", "yesterday") in the text.
  - Supertype: `uima.tcas.Annotation`
  - Features:
    - `time` (`uima.cas.String`): This feature holds the default time for the textual unit which is annotated by this annotation. The time string is expressed in the normalized ISO 8601 format (more specifically, it is a concatenation of the ISO 8601 calendar date and extended time: "YYYY-MM-DD hh:mm:ss").
- `EXCITEMENT.temporal.TemporalExpression`
  - Description: This type annotates a temporal expression, with a normalized time representation.
  - Supertype: `uima.tcas.Annotation`
  - Features:
    - `text` (`uima.cas.String`): This feature holds the original expression appeared on the text.

- `resolvedTime` (`uima.cas.String`): This feature holds the resolved time in ISO 8601 format. For example, "Yesterday", will be resolved into "2012-11-01", etc.

B.4. Types for Text Alignment

- `EXCITEMENT.alignment.AlignedText`
  - Description: This type represent an aligned textual unit. Its span refers to the "source" linguistic entity. This can be a token (word alignment), a syntax node (phrase alignments), or a sentence (sentence alignment).
  - Supertype: `uima.tcas.Annotation`
  - Features:
    - `alignedTo` (`uima.cas.FSArray`): This feature holds references to other AlignedText instances. The array can have multiple references, which means that it is one-to-many alignment. Likewise, a null array can also be a valid value for this feature, if the underlying alignment method is an asymmetric one; empty array means that this AlignedText instance is a recipient, but it does not align itself to other text.

Appendix C. Entailment Core Interfaces

C.1. `interface EDABasic` and related objects

C.1.1. `interface EDABasic`

```
public interface EDABasic
```

```
public interface EDABasic {

   // no constants

   // method signatures
   public void initialize(CommonConfig config);
   public TEDecision process(JCas aCas);
   public void shutdown();
   public void start_training(CommonConfig config);
}
```

C.1.2. `interface TEDecision`

```
public interface TEDecision
```

```
public interface TEDecision {

    // constants
    static final double CONFIDENCE_NOT_AVAILABLE = -1;

    public DecisionLabel getDecision();
    public double getConfidence();
    public String getPairID();
    public Object getInfo();
}
```

C.1.3. `enum DecisionLabel`

This section shows a possible implementation of hierarchical enum type of entailment decisions.

```
public enum DecisionLabel {
 Entailment(null),
 NonEntailment(null),
 Abstain(null),
 Paraphrase(Entailment),
 Contradiction(NonEntailment),
 Unknown(NonEntailment),
 //YourLabel(Paraphrase), // You can extend it like this...
 ;
```

```
 public boolean is(DecisionLabel e) {
  if (e == null) {
   return false;
  }
  for(DecisionLabel t = this; t != null; t=t.parent)
   if (e == t) {
    return true;
   }
  return false;
 }

 private DecisionLabel parent = null;
 private DecisionLabel(DecisionLabel parent) {
  this.parent = parent;
 }
}
```

C.2. interface EDAMulti*

C.2.1. interface EDAMultiT

```
public interface EDAMultiT
```

```
public interface EDAMultiT {

 // no constants

 // method signatures
 public List<TEDecision> processMultiT(JCas aCas);
}
```

C.2.2. interface EDAMultiH

```
public interface EDAMultiH
```

```
public interface EDAMultiH {

 // no constants

 // method signatures
 public List<TEDecision> processMultiH(JCas aCas);
}
```

C.2.3. interface EDAMultiTH

```
public interface EDAMultiTH
```

```
public interface EDAMultiTH {

 // no constants
```

```
 // method signatures
 public List<TEDecision> processMultiTH(JCas aCas);
}
```

C.3. class ModeHelper

```
public abstract class ModeHelper
```

```
public abstract class ModeHelper implements EDAMultiT, EDAMultiH {
 // fields
 private final EDABasic runner;

 // methods
 public abstract void setEDA(EDABasic e);
 public abstract List<TEDecision> processMultiT(JCas aCas);
 public abstract List<TEDecision> processMultiH(JCas aCas);
}
```

C.4. interface DistanceCalculation and related objects

C.4.1. interface DistanceCalculation

```
public interface DistanceCalculation
```

```
public interface DistanceCalculation {
 // no constants

 // methods signatures
 public void initialize(CommonConfig config);
 public DistanceValue calculation(JCas aCas);
}
```

C.4.2. class DistanceValue

```
public abstract class DistanceValue
```

```
public abstract class DistanceValue {
 // methods
 public double getDistance();
 public boolean getIsSimBased();
 public double getUnnormalizedValue();
 public Vector getDistanceVector();

 // fields
 private double distance;
 private boolean isSimBased;
 private double unnormalizedValue;
 private Vector distanceVector;
}
```

C.5. `class LexicalResource` and related objects

C.5.1. `class LexicalRule`

The class will be supplied as a concrete class. However, in this description, we are only outlining the object with an abstract class.

```
public abstract class LexicalRule<I extends RuleInfo> {

        private final String leftLemma;
 private final PartOfSpeech leftPos;
 private final String rightLemma;
 private final PartOfSpeech rightPos;
 private final I info;
 private final DecisionLabel relation;
 private final String originalRelation;
 private final String resourceName;
 private final double confidence;

 public String getLLemma();
 public String getRLemma();
 public PartOfSpeech getLPos();
 public PartOfSpeech getRPos();
 public I getInfo();
 public String getOriginalRelation();
 public DecisionLabel getRelation();
 public double getConfidence();

}
```

`DecisionLabel` is described in C.1. The following code outlines the PartOfSpeech.

C.5.2. `class PartOfSpeech`

```
public abstract class PartOfSpeech
{
 public abstract CanonicalPosTag getCanonicalPosTag();
 public abstract String getStringRepresentation();

 protected String posTagString;
 protected CanonicalPosTag canonicalPosTag;
}
```

C.5.3. `interface RuleInfo`

```
public interface RuleInfo {
        // This interface is an empty interface that serves as a
 // signature --- any class that implements this interface
 // will be used in LexicalRule as Info.
}
```

C.5.4. interface LexicalResource

```
public interface LexicalResource<I extends RuleInfo>
{
 List<LexicalRule<? extends I>> getRulesForRight(String lemma,
      PartOfSpeech pos);
 List<LexicalRule<? extends I>> getRulesForRight(String lemma,
      PartOfSpeech pos, DecisionLabel l);

 List<LexicalRule<? extends I>> getRulesForLeft(String lemma,
      PartOfSpeech pos);
 List<LexicalRule<? extends I>> getRulesForLeft(String lemma,
      PartOfSpeech pos, DecisionLabel l);

 List<LexicalRule<? extends I>> getRules(String leftLemma,
      PartOfSpeech leftPos, String rightLemma, PartOfSpeech rightPos);
 List<LexicalRule<? extends I>> getRules(String leftLemma,
      PartOfSpeech leftPos, String rightLemma, PartOfSpeech rightPos, DecisionLabel l);
}
```

C.6. class SyntacticResource and related objects

C.6.1. class SyntacticRule

We are expecting the `SyntacticRule` as a concrete class, but here we are representing them with an abstract class, since the description is partial.

```
public abstract class SyntacticRule {

       // Actula BasicNode will be a class that is
       // parameterized by generics.
       // Here They are written as a simple name.

       public BasicNode getRightHandSide();
       public BasicNode getLeftHandSide();
       public BidrectionalMap<BasicNode,BasicNode> getMapNodes();

       protected BasicNode leftHandSide;
       protected BasicNode rightHandSide;
       protected BidirectionalMap<BasicNode,BasicNode> mapNodes;

}
```

C.6.2. class BasicNode and related classes

The abstract classes described in this section outlines the content of a basic node as a Java object.

```
public final class BasicNode extends AbstractNode<Info, BasicNode>
{
 public BasicNode(Info info)
 {
```

```
  super(info);
 }
}
```

```
public abstract class AbstractNode<T,S extends AbstractNode<T,S>>
{
 protected AbstractNode(T info)
 {
  this.info = info;
 }

 public T getInfo()
 {
  return info;
 }

 public void addChild(S child)
 {
  if (this.children==null)
   this.children = new ArrayList<S>();

  this.children.add(child);
 }

 public ArrayList<S> getChildren()
 {
  return children;
 }


 public boolean hasChildren()
 {
  if (children!=null)
  {
   if (children.size()>0)
   {
    return true;
   }
  }
  return false;
 }

 protected T info;
 protected ArrayList<S> children;
}
```

The above two classes outlines the `BasicNode` as a tree node. Actual content of the node is represented with a set of info objects. The following two class outlines `info` and `DefaultInfo`

```
public interface Info extends Serializable
{
 public NodeInfo getNodeInfo();
 public EdgeInfo getEdgeInfo();
 public String getId();
```

```
}
```

```
public abstract class DefaultInfo implements Info
{

 public DefaultInfo(String id, NodeInfo nodeInfo, EdgeInfo edgeInfo)
 {
  this.id = id;
  this.nodeInfo = nodeInfo;
  this.edgeInfo = edgeInfo;
 }

 public NodeInfo getNodeInfo()
 {
  return this.nodeInfo;
 }

 public EdgeInfo getEdgeInfo()
 {
  return this.edgeInfo;
 }

 public String getId()
 {
  return this.id;
 }

 protected NodeInfo nodeInfo;
 protected EdgeInfo edgeInfo;
 protected String id;
}
```

The following class outlines `NodeInfo`.

```
public interface NodeInfo
{
 public String getWord();
 public String getWordLemma();

 public SyntacticInfo getSyntacticInfo();
 public NamedEntity getNamedEntityAnnotation(); // returns simple enum.

 public boolean isVariable();
 public Integer getVariableId();
}
```

The following class outlines `EdgeInfo`.

```
public interface EdgeInfo
{
 public DependencyRelation getDependencyRelation();
```

```
}
```

The following classes outlines data classes that are used in the `NodeInfo` and `EdgeInfo`. They are expected to be concrete classes. However, here they are drawn as abstract, to show only the relevant data part.

```
public abstract class DependencyRelation {
        public String getStringRepresentation();
        public DependencyRelationType getType(); // returns an enum type

 protected String itsStringRepresentation; // raw output from parser
        protected DependencyRelationType type = null; // canonical relation
}
```

```
public interface SyntacticInfo {
        public PartOfSpeech getPartOfSpeech();
}
```

C.6.3. `interface SyntacticResource`

```
public interface SyntacticResource {
        List<RuleMatch> findMatches(BasicNode currentTree);
}
```

```
public abstract class RuleMatch {

        public BasicNode getTarget();
 public SyntacticRule getRule();

        protected BasicNode target;
        protected SyntacticRule rule;

}
```

Appendix D. Supported Raw Input Formats

Note that only RTE-5 format is mandatory (EDAs should support them), and other formats are optional (EDAs may support them). Also, the following DTD's all have a minor modification of `lang` attribute. See Section 5.2, “Input file format ”.

RTE-1 (XML):

```
<!ELEMENT entailment-corpus (pair+)>
<!ATTLIST entailment-corpus
lang CDATA #IMPLIED> <!-- ISO 639-1 code like "EN", "DE", "IT" -->
<!ELEMENT pair (t,h)>
<!ATTLIST pair
id CDATA #REQUIRED
task (CD|MT|PP|RC|IR|IE|QA) #REQUIRED
value (TRUE|FALSE) #REQUIRED >
<!ELEMENT t (#PCDATA)>
<!ELEMENT h (#PCDATA)>
```

RTE-2 (XML):

```
<!ELEMENT entailment-corpus (pair+)>
<!ATTLIST entailment-corpus
lang CDATA #IMPLIED> <!-- ISO 639-1 code like "EN", "DE", "IT" -->
<!ELEMENT pair (t,h)>
<!ATTLIST pair
id CDATA #REQUIRED
entailment (YES|NO) #REQUIRED
task (IR|IE|QA|SUM) #REQUIRED >
<!ELEMENT t (#PCDATA)>
<!ELEMENT h (#PCDATA)>
```

RTE-3 (XML):

```
<!ELEMENT entailment-corpus (pair+)>
<!ATTLIST entailment-corpus
lang CDATA #IMPLIED> <!-- ISO 639-1 code like "EN", "DE", "IT" -->
<!ELEMENT pair (t,h)>
<!ATTLIST pair
id CDATA #REQUIRED
entailment (YES|NO) #REQUIRED
task (IR|IE|QA|SUM) #REQUIRED
length (LONG|SHORT) #REQUIRED >
<!ELEMENT t (#PCDATA)>
<!ELEMENT h (#PCDATA)>
```

RTE-4 (XML):

```
<!ELEMENT entailment-corpus (pair+)>
<!ATTLIST entailment-corpus
```

```
lang CDATA #IMPLIED> <!-- ISO 639-1 code like "EN", "DE", "IT" -->
<!ELEMENT pair (t,h)>
<!ATTLIST pair
id CDATA #REQUIRED
entailment (ENTAILMENT|CONTRADICTION|UNKNOWN) #REQUIRED
task (IR|IE|QA|SUM) #REQUIRED >
<!ELEMENT t (#PCDATA)>
<!ELEMENT h (#PCDATA)>
```

RTE-5 (XML):

```
<!ELEMENT entailment-corpus (pair+)>
<!ATTLIST entailment-corpus
lang CDATA #IMPLIED> <!-- ISO 639-1 code like "EN", "DE", "IT" -->
<!ELEMENT pair (t,h)>
<!ATTLIST pair
id CDATA #REQUIRED
entailment (ENTAILMENT|CONTRADICTION|UNKNOWN) #REQUIRED
task (IR|IE|QA) #REQUIRED >
<!ELEMENT t (#PCDATA)>
<!ELEMENT h (#PCDATA)>
```