EXCITEMENT EXploring Customer Interactions through Textual E

EXCITEMENT open platform: Architecture and Interfaces

draft

10 March 2014

Authors:

Tae-Gil Noh, Heidelberg University <noh@cl.uni-heidelberg.de>
Sebastian Pado, Heidelberg University <pado@cl.uni-heidelberg.de>
Asher Stern, Bar Ilan University <astern7@gmail.com>
Ofer Bronstein, Bar Ilan University <oferbr@gmail.com>
Rui Wang, DFKI <wang.rui@dfki.de>
Roberto Zanoli, FBK <zanoli@fbk.eu>

Abstract:

This specification defines various parts of the EXCITEMENT open platform.

Table of Contents

Appendixes

1. Introduction

1.1. Preface to the specification version 2.0

Short history of the specification. EXCITEMENT open platform specification has been first published in June 2012. This first version defined backbones of EXCITEMENT open platform with major component types, data structures and conventions for the development of the open platform. The specification has been updated a few time with the actual development progress. Version 1.1 was released in August 2012. This version was a major update that provided missing parts to version 1.0, and became the blueprint for the platform development over 2012 and 2013. The last version of specification 1.1 is specification version 1.1.4 [http://hltfbk.github.io/Excitement-Open-Platform/specification/spec-1.1.4.pdf], published in October 2013. EXCITEMENT open platform (EOP) [http://hltfbk.github.io/Excitement-Open-Platform/] has been successfully developed following the specification. EXCITEMENT open platform (version 1.1.1, at the time of writing) is a matured, well-documented, multi-lingual, open-source Textual Inference platform.

Scope of the specification 2.0 . Specification 1.0 introduced the common platform that can support different RTE (Recognizing Textual Entailment) systems on multiple languages. One of the main goal was (and still is) making the platform a paradigm-neutral platform, where various Textual Inference approaches can share their components. This goal was very broad, and hard to achieve in general. Thus, the consortium adopted a practical strategy of analyzing and utilizing three existing RTE systems as the basis. This was a very successful strategy, and delivered a multilingual platform with easy-to-share components. But the strategy also had its limits. For example, the specification did not cover some other RTE approaches (like alignment-based method), and its scope did not included support for novel research topics. In this sense, spec 1.0 was an enabler that makes the fusion of existing three target RTE systems into one coherent platform. We now try to broaden the scope of the open specification with the version 2.0 and beyond. Basically, specification 2.0 is still a linear extension of specification version 1.1. All the major structures, and components are already stable and there is no major updates on those existing modules. However, Spec 2.0 also aims to achieve more than consolidating existing pre-EOP systems. Specification 2.0 adds contents to support additional data types and component types that were not part of any of the existing systems. This includes alignment components, phrases-level resources, context sensitive lexical resources, and support for new Entailment Decision Algorithms (EDAs). The following list outlines major new component types and updated sections that are new to specification 2.0.

- Context sensitive Lexical resource (Section 4.11, “Context Sensitive Lexical Resource”)
- Phrase level knowledge resource (TODO add XREF once added)
- Annotation component (Section 4.13, “Annotator component”)
- Alignment component and alignment CAS types (Section 4.14, “Alignment component”)
- Training interface for components (Section 4.15, “Interface `Trainable`”)
- Sentence Similarity Decision Algorithm (Section 4.16, “Semantic Similarity Decision Algorithm”)

Please note that other sections, such as LAP interface sections, configuration sections and details of syntactic resources, also got minor updates. For the full list of changes since specification version 1.0 (Deliverable 3.1a), please see the Appendix Appendix E, *List of changes per version* .

Related documents. If you are reading this specification for the first time (no prior experience on the specification or the platform), maybe it is better to start with a journal paper that describes the platform in general [JNLE2013] . Also, UIMA part of the platform (type systems and UIMA module adaptation) is well summarized in a workshop paper [UIMAWS2013].

1.2. Introduction to EXCITEMENT open specification

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.

EXCITEMENT has two primary goals (see the project proposal for details):

1. Goal A: Develop and advance a comprehensive open-source platform for multi-lingual textual inference, based on the textual entailment paradigm.

2. Goal B: Inference-based exploration and processing of customer interactions at the statement level, for multiple languages and interaction channels.

where Goal B builds strongly on the development of the platform for Goal A. We envisage the role of this platform to be similar to the one played by MOSES in the Machine Translation community -- that is, as a basis for reusable development. For these goals, we work towards 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) [BIUTEE].

- *EDITS*, an edit distance-based approach to textual entailment recognition (FBK) [EDITS].

- *TIE*, Textual Inference Engine (DFKI) [TIE].

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)

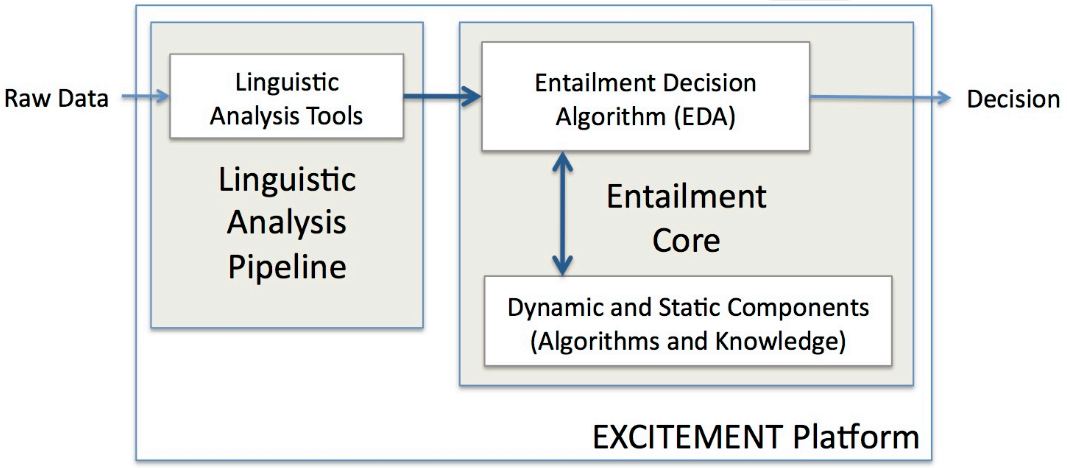
Outside the scope of this document is the specification of the transduction layer. This layer is part of the work towards Goal B. It translates between the queries posed by the industrial partners' use cases and entailment queries that can be answered by the open platform developed for Goal A. The transduction layer is described in a separate document (Deliverable 3.2b).

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


Raw Data
Linguistic Analysis Tools
Linguistic Analysis Pipeline
Entailment Decision Algorithm (EDA)
Entailment Core
Dynamic and Static Components (Algorithms and Knowledge)
EXCITEMENT Platform
Decision

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 endeavor, 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.

2.2. Engines: Instantiations of the Platform

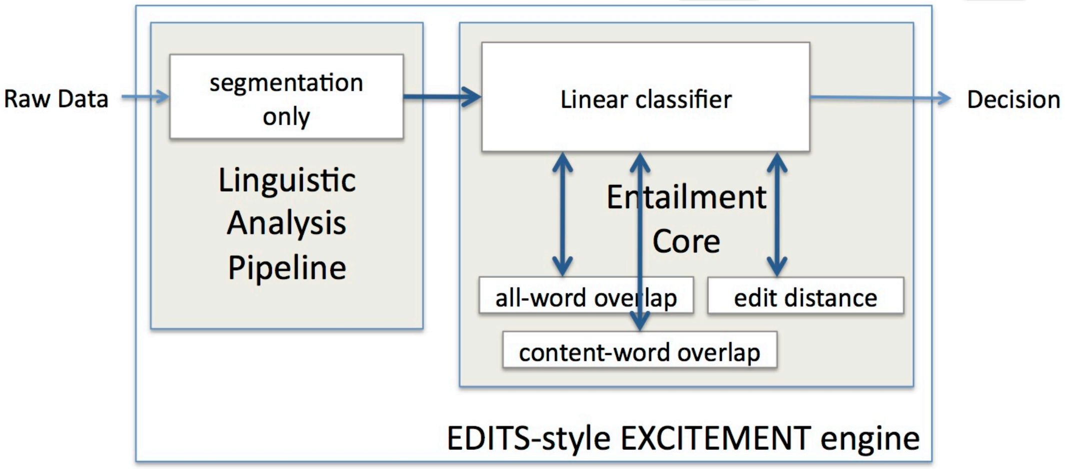
In order to use the EXCITEMENT infrastructure, a user will have to configure the platform for his application setting, that is, for his language and his analysis tools (on the pipeline side) and for his algorithm and components (on the entailment core side).

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



2.3. Concepts and Terminology

2.3.1. 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.

2.3.2. 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 instance 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 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 re-usability, 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 data-set.

2.3.3. 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.
- *Linguistic formalism independence.* The data structure should be independent of specific linguistic theories and formalisms.
- *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 module to already existing pipelines.

On the top level, the unification of UIMA components is achieved on two levels.

1. 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 code. 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.

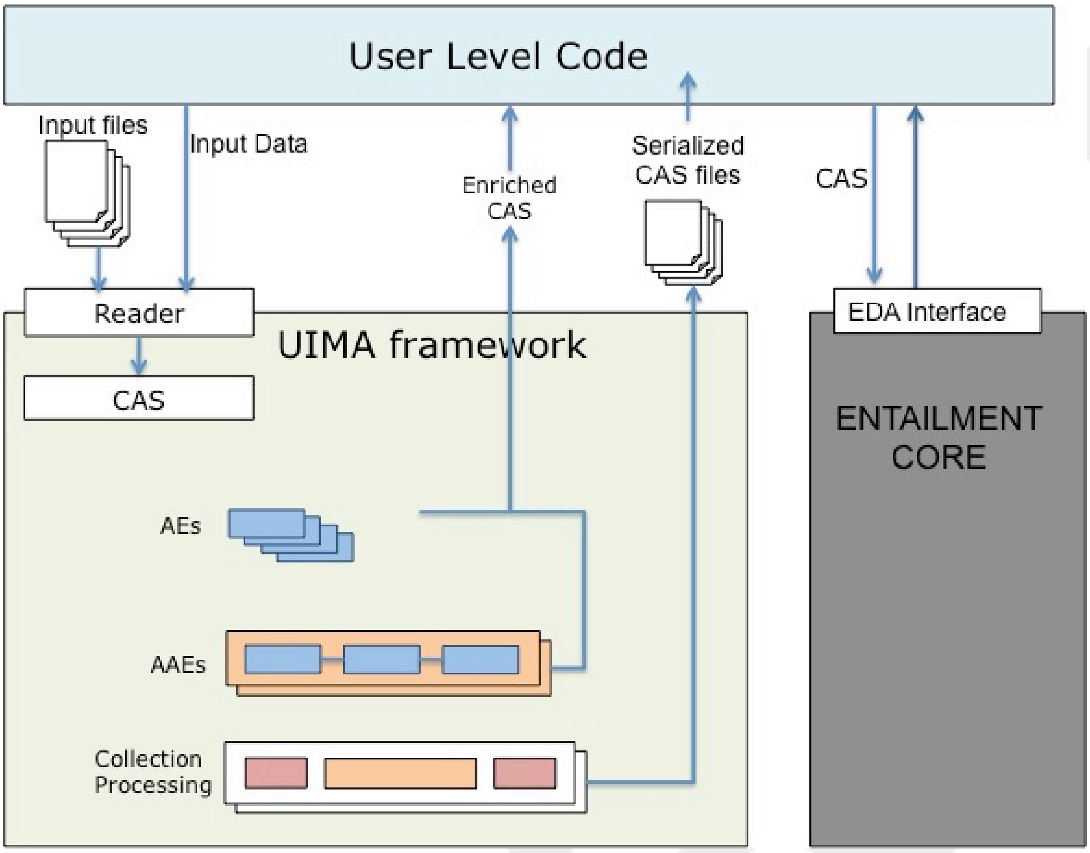
2. The second unification is done with common data representation. If the inputs and outputs of each component are not compatible, then the unification of components' behaviors is not really meaningful. In UIMA, the compatibility of input and output is provided by the Common Analysis Structure (CAS) [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 the result of a POS tagging step), and what it will add on the CAS after the processing (like a 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. 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 share a set of contracted methods, and they will be accessed by the UIMA framework. All analysis results of an 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 calls of a sentence splitter AE, a POS tagger AE, and a parser AE. UIMA provides a standard way to compose analysis pipelines without changing anything on the individual analyzers.

Collection Processing (and UIMA Runtime). A 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 on the same dataset. In this case, repeating the same linguistic preprocessing would be redundant. AAEs do not provide any capability for accessing files or processing collections. It is the responsibility of the AAEs caller to recognize such cases and avoid redundant calls. To guarantee that this happens in a uniform manner, UIMA provides the concepts of *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. If a user calls an AE directly (via the framework), the component is actually running in the caller's space (same process, thread). In this case, running AEs on multiple files would still be the caller's 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 the UIMA community provide various runtimes, from simple single thread runtime to asynchronous distributed runtimes. Examples for UIMA runtimes are UIMA CPE/CPM (collection processing engine / collection processing manager), UIMA AS (Asynchronous scale-out), UIMAfit simple pipeline, and the UIMAfit hadoop runtime.

The LAPs of EXCITEMENT exist primarily for processing textual entailment data. As a secondary goal, the LAPs need to provide linguistic analysis capabilities for the user level code and the transduction layer. With these goals in mind, it is possible to formulate more focused requirements for EXCITEMENT LAP components. An EXCITEMENT LAP should provide the following capabilities:

- Calling individual analysis components: A single AE represents an individual linguistic analysis component, like tagger, tokenizer, or parser. The user code can call an AE, or a set of AEs to utilize the analysis capability provided by each component.

- Calling a prepared pipeline: A specific EDA needs a specific level of analysis. For example, one EDA might need "tokenizer - POS tagger" analysis sequence, and another EDA might need "tokenizer - POS tagger - parser - NER" sequence. UIMA AAE can provides this specific analysis sequence that is optimized for a specific EDA. Note that AAEs do not (and should not) know about the concepts of Text and Hypothesis. AEs and AAEs treat given CAS (or CAS view) as plain running text, and simply process a sequence of analysis.

- Calling a collection processing for EDAs: The EDA interface requires a CAS that is prepared for EDAs. This will include types and views specially designed to represent textual entailment problems. AEs and AAEs do not know about TE-related types or setups. Such entailment specific types are prepared by a "collection processor". The flow of a collection processor starts with a "collection reader". A collection reader reads one (or more) textual entailment problem input (like RTE data), and generates a UIMA CAS with views and types defined for entailment problems. Then, the collection processor calls one configured AAE for each view. For example, if the processor calls an AAE prepared for some EDA (say EDA-A), the CAS is now ready to be processed by "EDA-A". If the views are processed by an AAE prepared for "EDA-B", the resulting CAS (or CASes) is prepared for "EDA-B".

3.2.3. Adoption of UIMA in EXCITEMENT

At the time of writing of this specification (V1.0), the EXCITEMENT consortium has made the following decision regarding the adoption of UIMA within the linguistic analysis pipeline:

- We will definitely adopt the UIMA type system.

- Given the existence of working pipelines that form part of the three existing systems, we do not immediately adopt a stance with regard to the adoption of the UIMA runtime system. The minimal level of interoperability that these pipelines must provide at the moment is that they conform to the LAP input (Section 5.2, “Input file format ”) and output (CAS) specifications.

- For the future (2013/14), we see a number of graded options regarding UIMA adoption:

  - Status quo (minimal interoperability)

  - Wrapping of pipelines into AAEs (basic use of UIMA runtime)

  - Decomposition of pipelines into AEs (advanced use of UIMA runtime)

- The choice among these future strategies will be taken when consortium partners have had more practical experience in working with UIMA.

More formally, the following requirements fall out of these decisions:

- All pipeline implementations **must** utilize the common type system. The common type system is described in Section 3.3, “Type Systems”. The EDA implementer **may** extend the type system to meet particular information needs. This is discussed in Section 3.3.5, “Extending existing types”.

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

- It is **recommended** that the implementer provides the ability to pre-process a set of documents and produce a set of properly serialized CAS structures that can be consumed by the EDA. This is specified in Section 3.4.3, “Providing serialized outputs for a set of documents”.

- The implementer **can** 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, “An alternative: Independent pipelines with CAS output”.

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

Types needed for EXCITEMENT will be defined based on the existing types of the UIMA community. For generic linguistic analysis results, we adopt many types from DKPro. The UIMA types of DKPro components have been used in many applications, and they are relatively well matured types. We also add a few new types that were absent in DKPro, like temporal annotations and semantic role labels. The adopted and proposed UIMA types for generic linguistic analysis are described in Section 3.3.3, “Types for generic NLP analysis results”, and their type listings are included in Appendix A, *Type Definition: types for general linguistic analysis*.

This specification also defines some UIMA types that are designed to represent textual entailment problems and related annotations. 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 are 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 constitute a relatively matured system, since it has been around for some years.

3.3.1. Relevant UIMA concepts: Artifact, View, SOFA

Artifact is a UIMA concept which describes the raw input. For example, a document, a web page, or a RTE test dataset, etc, are all artifacts. For the moment, let's assume that we have a web page 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, an 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 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.TOP`: This type is the top type of the CAS type hierarchy. All CAS types are subtypes of `TOP`. It does not have any feature.

- `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.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.DocumentAnnotation`: 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

This section outlines adopted and proposed UIMA types for generic NLP analysis results. The full list of the types are given 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 information, including the 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.Annotation`). 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.

Also note that the POS type can be extended to cover other (more detailed, or domain-specific) tagsets. See also Section 3.3.5, “Extending existing types”.

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. Constituency parse result

Constituency parsing results can be represented with `DKPRO.syntax.type.constituent`. It has a set of types related to constituency parsing. `constituent` type represent a parse node, and holds required information such as `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.*`).

Note that the EXCITEMENT platform has adopted dependency parse trees as its canonical representation for syntactic knowledge (as described in Section 4.8, “Interface of syntactic knowledge components ”). EDA implementers can use constituency parsers and their results within their code, but constituency parse trees cannot be used together with EXCITEMENT syntactic rule-bases, unless they are converted to dependency parse trees.

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 a 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, EXCITEMENT.semanticrole.Predicate and EXCITEMENT.semanticrole.Argument.

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 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 EXCITEMENT.semanticrole.Argument. It holds the predicate's arguments.

Argument is a uima.tcas.Annotation. It represents an argument. It has two features; the argument's semantic role (type), and a backward reference to the predicate that governs the argument. They are:

- argumentName (uima.cas.String): This feature represents the semantic role (type) of this argument. The fact that this feature is a String means that arbitrary role labels can be adopted, such as the PropBank argument set (A0, A1, AM-LOC, etc.), the FrameNet role set (Communication.Communicator, Communication.Message, etc.), or any other.

- predicates (uima.cas.FSArray): This feature is an array of EXCITEMENT.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. it is easier to find the corresponding predicates from each argument).

3.3.3.9. Types for Temporal Expression

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

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 appearing in 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.

temporal.TemporalExpression has four subtypes, which closely reflects TIMEX3 classification [TIMEX3] of temporal expression. They are: temporal.Date, temporal.Time, temporal.Duration, and temporal.Set.

3.3.3.10. 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.

- alignmentType (uima.cas.String): This feature gives additional information about the alignment ("type" of the alignment) as a string. The feature can be null, if no additional information is available.

The alignedTo feature is an array that can contain 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.

3.3.4. Additional Types for Textual Entailment

This section outlines the types proposed for textual entailments and related sub-tasks. Full listing of the mentioned types are listed in Appendix B, *Type Definition: types related to TE tasks* .

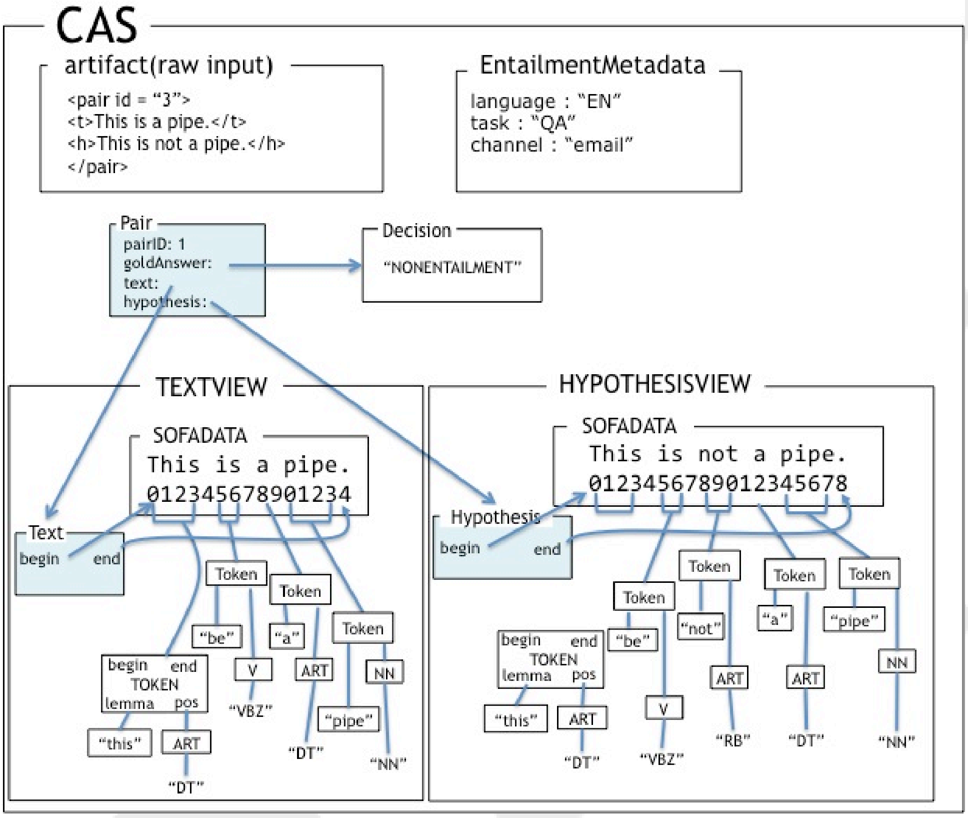
3.3.4.1. Representation of Text and Hypothesis

The basic approach of the platform is to use a single CAS to represent a single entailment problem. The CAS holds text, hypothesis and associated linguistic analysis annotations.

Let's first consider Figure 4, “Example of an entailment problem in CAS” which shows an example of entailment problem represented in a. It includes UIMA CAS elements like artifact, view, and SOFA (cf. Section 3.3.1, “Relevant UIMA concepts: Artifact, View, SOFA”). The original raw input is shown in the figure as the artifact. It is a simple XML annotation with a T-H pair. The artifact contains two plain-text subjects of analysis (SOFA). Analysis tools like taggers will only see this SOFA data, instead of the artifact XML. All annotations (all types that inherits uima.tcas.Annotation) are annotated on top of the SOFA. For example, the first token annotation of the text ("this") is annotated on TextView SOFA position 0 to position 3. This SOFA positional span underlies the most common way of accessing fetching UIMA text annotations, namely to retrieve, e.g., "all annotations of type Token from the TextView SOFA span 0 - 14".

Each view has exactly one SOFA. In this example, each view is enriched by token annotations and POS annotations. Both token and POS are annotation types, and have `begin` and `end` fields, which shows the span of the annotation. (POS types also have begin and end span. They are omitted here for clarity.)

Figure 4. Example of an entailment problem in CAS



The figure also shows a set of entailment specific annotation types: they are `entailment.Pair`, `entailment.Text`, `entailment.Hypothesis`, and `entailment.EntailmentMetadata`. The example has 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".

The following subsections define the annotations in detail.

3.3.4.1.1. Type `entailment.EntailmentMetadata`

`EXCITEMENT.entailment.EntailmentMetadata` provides metadata for each entailment problem. It is an extension of `uima.cas.TOP`. An instance of this type should be indexed directly on the CAS (not on `TextView` or `HypothesisView`).

- `language (uima.cas.String)`: this string represents the language of the CAS and its entailment problem. It holds two character language identifier, as defined in [ISO 639-1]).
- `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 "automatic transcription", "manual 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.

- `TextDocumentID (uima.cas.String)`: This field can hold a string that identifies the document of the `TextView`. This feature must have a value, if `TextCollectionID` is not null.

- `TextCollectionID (uima.cas.String)`: This field can hold a string that identifies the collection name where the document of the `TextView` belongs to.

- `HypothesisDocumentID (uima.cas.String)`: This field can hold a string that identifies the document of the `HypothesisView`. This feature must have a value, if `HypothesisCollectionID` is not null.

- `HypothesisCollectionID (uima.cas.String)`: This field can hold a string that identifies the collection name where the document of the `HypothesisView` belongs to.

The values of these fields are unrestricted Strings. Pipeline users are responsible for documenting the semantics of the values if they want to ensure interoperability.

3.3.4.1.2. Type `entailment.Pair`

Text-Hypothesis pairs are represented by the type `EXCITEMENT.entailment.Pair`. The type is a subtype of `uima.cas.TOP`. An instance of `Pair` should be indexed directly on the CAS (not on `TextView` or `HypothesisView`). 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.

3.3.4.1.3. `entailment.Text`, `entailment.Hypothesis` and `entailment.Decision`

`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 does not have any features. A `Text` instance is always referred to by a `Pair` instance.

Note that `Text` annotations and `TextView` are two different concepts. A `TextView` is a container that contains a SOFA plus all of its annotations, including `entailment.Text` Thus, the fact that a sentence forms part of a `TextView` does not automatically mean that it is compared to a hypothesis. This only happens if it is annotated with `entailment.Text` and linked with `entailment.Pair`.

The situation is similar for `EXCITEMENT.entailment.Hypothesis`. It 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 does not have any features either. A `Hypothesis` instance is always referred to by a `Pair` instance. `Hypothesis` annotations and `HypothesisView` are different entities, and sentences are only compared to a specific text annotation when they are annotated with `entailment.Hypothesis`, like for texts.

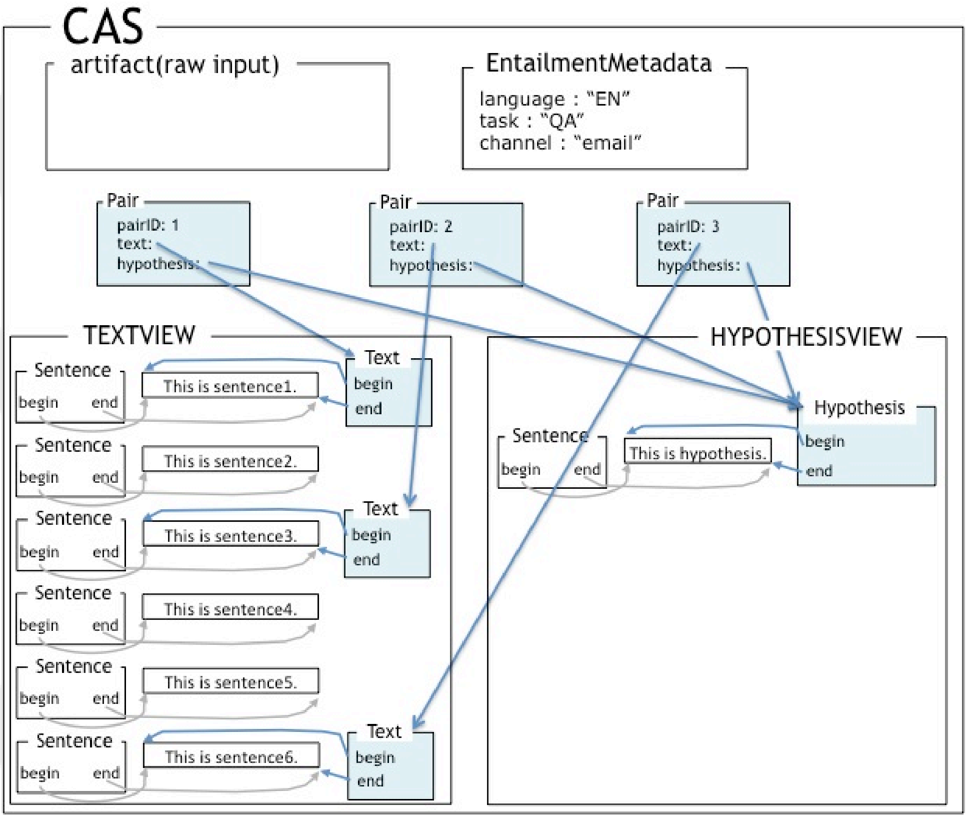
The EDA `process` interface receives CAS data as a JCas object (the UIMA-created Java type hierarchy corresponding to the UIMA CAS types). CASes that feed into the EDA `process` interface must have at least a single `entailment.Pair`. Naturally, each view (`TextView` and `HypothesisView`) must have at 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). This UIMA type has a corresponding Java enum type within the entailment core (see Section 4.2.1.6, “enum `DecisionLabel`”).

3.3.4.2. Generalization to multiple pairs

Figure 5, “Example of an entailment problem with multiple pairs” shows another example of a CAS with entailment types. In this figure, the CAS has multiple pairs. The TextView holds a document with six sentences. All six sentences are annotated by sentence splitter, and additional analyzers (for clarity, only sentence annotations are shown). Among the analyzed sentences, the `entailment.Text` annotations are marked on the 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 regarded 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 tokens in the context sentences to get better results. (Note that generic components like tokenizer, parser, and co-reference resolver, do not aware of entailment specific annotations, and process all data in the given view. "Generic" verses "TE specific" LAP component is described in the next subsection Section 3.3.4.3, “Two groups of LAP components”.)

Figure 5. Example of an entailment problem with multiple pairs



Note that EDAs will only decide entailment of pairs that are explicitly marked with `Pair` annotations. Existence of text or hypothesis annotation does not automatically lead to processing in an EDA. For example, if the pair-2 is missing from the figure, the `Text` annotated on sentence-3 will not be compared to the hypothesis sentence. Also note that nothing stops us from annotating a text item that overlaps with other text items. For example, it would be a legal annotation if we add a new text item that covers from sentence-2 to sentence-4 (which will include the text of pair-1).

3.3.4.3. Two groups of LAP components

All EXCITEMENT LAP components can be divided into two groups:

- *generic analysis components*
- *TE specific components*

The most important difference between the two groups is awareness of the two views (the `TextView` and the `HypothesisView` as introduced in Section 3.3.4.1.3, “`entailment.Text`, `entailment.Hypothesis` and `entailment.Decision` ”). Only the second group knows about the views, and also the entailment specific annotations (like `entailment.Metadata`). General LAP components (like taggers and parsers) only process a view. Keeping generic analysis tools as single view annotators is intentional and well-established UIMA practice. It enables the tools as generic as possible. In our case, user level or transduction layer can call analysis components not only on TE problems, but also on any textual data.

Pipelines with two (or more) views use generic AEs to annotate one of its views. If a pipeline needs to annotate all of its views, it needs to call the generic LAP component multiple times. For instance, a typical pipeline that produce a CAS for a EDA looks like this:

1. collectionReader reads a TE problem from the text format (like that of Section 5.2, “Input file format ”)
2. collectionReader generates a CAS with two views, entailment.metadata, and entailment.pair.
3. runtime calls a generic AE (or AAE) on the Text View (for example, a Tagger+Parser AE/AAE)
4. runtime calls a generic AE (or AAE) on the Hypothesis View (same Tagger+Parser AE/AAE)
5. runtime calls a TE specific AE on the CAS (e.g. a TE-specific alignment annotator)
6. the CAS is ready and passed to the entailment core.

Note that generic analysis annotators are called twice for once on TextView and once on HypothesisView (step 3 and 4), while TE-specific tool (step 5) is called only once. Also note that this is only one example, and a whole set of different processing pipelines can be designed. For example, to process a big set of short texts, we can imagine a pipeline that first call taggers for all problems as a single document and then generate multiple CASes for efficiency reasons, etc.

3.3.4.4. Other cases: T-H from multiple documents, T-H on a same document

Ordinary annotators (taggers, parsers, etc) do not (and should not) know about the views of `TextView` and `HypothesisView`. They only see one view at a time, its SOFA as a single document. Thus the two views of T and H naturally represent two separate documents, one for text and one for hypothesis.

One problem of this setup is that we have to process other cases like:

- Text (or Hypothesis) from Multiple documents.
- Text and Hypothesis on the same document.

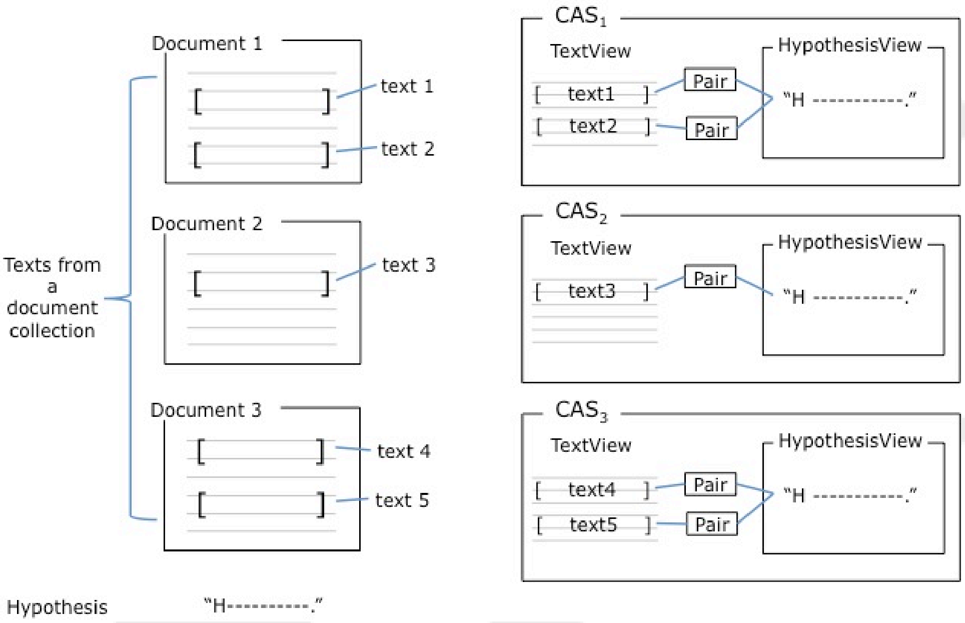
The following subsections describe how we will represent such cases with the proposed types and CAS.

3.3.4.4.1. Multiple document problem: decomposition to multiple CASes

Some problems involve multiple documents. Examples are the main task of RTE6 and RTE7. They have TE problem that consists of multiple texts (T) and a single hypothesis. The texts are scattered over a set of documents (collection). Additionally, there are global features associated with the document collection (e.g., topic).

We can break down this case with multiple CASes with the proposed CAS representation. The following figure shows one artificial example. In the figure, it has a single hypothesis and multiple texts. The texts are located in three different documents. It is possible to represent this problem with three CASes.

Figure 6. Multiple document TE problem decomposed into multiple CASes.



Each CAS of the decomposed one will look like Figure 5, “Example of an entailment problem with multiple pairs”. Note that the multi document example is decomposed into several CASes whose number corresponds to the number of documents. Each view holds a single document in this setup, and normal annotators can process each view just as a normal document.

`EXCITEMENT.entailment.EntailmentMetadata` holds two additional features for multiple document cases. It has `documentID` and `collectionID` for both text and hypothesis. Document ID identifies the document assigned in the view, and collection ID can show the collection where the document is belong to.

Note that this decomposition assumes that breaking down the multiple document case into multiple single document problem does not change the final output. See Section 3.3.4.4.3, “More complex problems and their use-cases ” for more complex cases of multiple text/hypothesis.

3.3.4.4.2. T and H on same document : Two views hold the same document

Since we have two separate views for text and hypothesis, it can be problematic when we have the text and the hypothesis from the same document.

In such a case, we will have two views with the same document (two views have the same document with same analysis result). All other TE related annotations stays normal, like metadata and entailment problem annotation. Only document specific annotations (like tagging, parsing) are duplicated into two views.

Note that this "duplicated analysis result in two views" does not mean the analysis of the document has to be done twice. We can setup a special pipeline for single document T-H case, where it first runs analysis and copies the result into two views.

3.3.4.4.3. More complex problems and their use-cases

Note that the above two mappings (multiple documents THs and single TH) simplify EDA interfaces. For EDAs, all inputs look alike with two views with `entailment.Pair` type annotations. However, we can imagine more complex use cases where no trivial mapping is possible. Within the WP3 discussions, the following tasks have been discussed for possible future support.

- Text expansion: Automatically generating a (large) set of additional texts that can entail the given input text.
- Dialog system: Ranking hypothesis for the given text. The goal is to rank, instead of decision.
- Hypothesis with variables: Hypothesis that includes variables, where the variables can be replaced by parts (like words) that appears in the text.

For such cases, we will probably additional interfaces on EDA side, also with new problem descriptions. However, this version of specification only deals with existing, well-known RTE problem sets. The WP3 members have agreed that additional use-cases will only be discussed in a future version of the specification.

3.3.4.5. Types for Predicate Truth

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

We need four annotation types according to the annotations that are needed for predicate truth annotator. They are `PredicateTruth`, `ClauseTruth`, `NegationAndUncertainty`, and `PredicateSignature`. 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.PredicateSignature` is a `uima.tcas.Annotation`. It represents an implication signature. It has the following feature:

- `value` (`PredicateSignatureValue`, a string subtype): This represents the value of the annotation. The value type is a subtype of string permitting only the values "+/-", "+/?", "?/-", "-/+", "-/?", "?/+", "+/+", "-/-", "?/?".
- The implementation uses a set of more fine grained values, that may grow in the future. `PredicateSignatureValue` will be extended and modified accordingly

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 would be ideal if each linguistic analysis module (like POS taggers, parsers or coreference resolvers) were provided as a generic individual UIMA analysis engine (AE). With individual AEs, user code would be able to call these modules as UIMA components, as described in Section 3.2.2, “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 within 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.

In some cases, one may choose NOT to provide individual analysis components as UIMA components. See Section 3.4.4, “An alternative: Independent pipelines with CAS output” for such a 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 serialized outputs for a set of documents

Pre-processing of a set of document (collection) is often needed for NLP applications. In textual entailment context, a pre-processed set of training/testing data can be beneficial for various cases.

EDA implementers **should** provide the collection pre-processing capability.

- 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. An alternative: Independent pipelines with CAS output

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 situations where EDAs depend on specialized analysis modules which form a pipeline that cannot easily be divided into individual UIMA analysis engines. For example, assume that there is a parser and a coreference resolution system 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 these cases, the developers should provide the pipeline as an 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.

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.

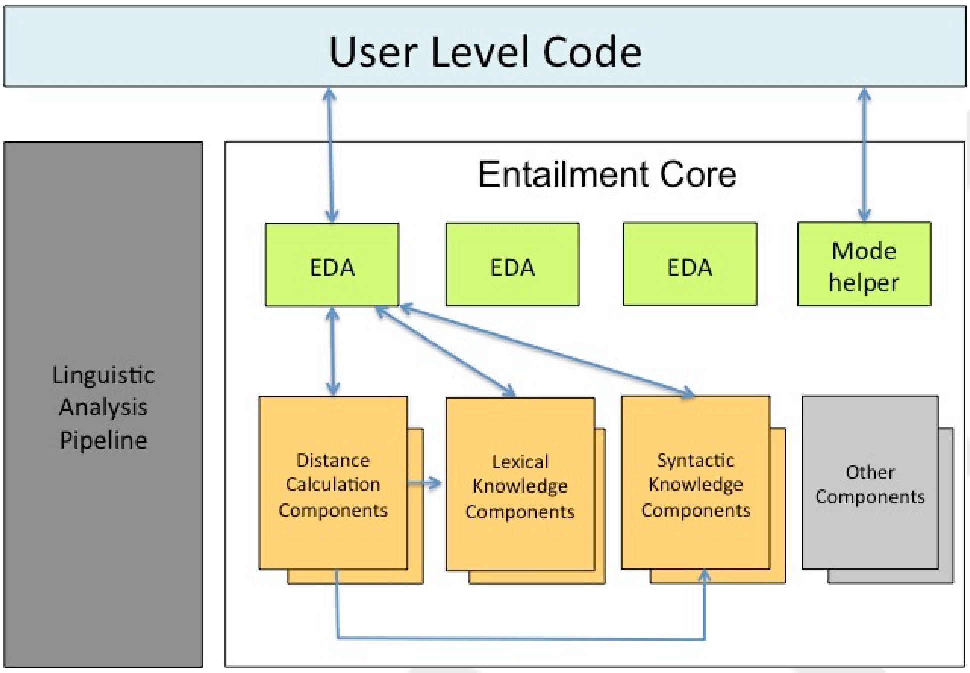
- *Further extensibility.*Users should be able to extend the set of components beyond the standard classes in this specification. This may however involve more effort.

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. details of their training regimens) are not be specified but in the scope of the component's developer

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 7. Common components within EXCITEMENT entailment core.


User Level Code
Linguistic Analysis Pipeline
Entailment Core
EDA
EDA
EDA
Mode helper
Distance Calculation Components
Lexical Knowledge Components
Syntactic Knowledge Components
Other Components

The top-level interfaces of the entailment core consist of the EDA interface and the mode helper interface. EDAs are the access points for entailment decision engines. They receive annotated input in the form of CAS, and return 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, “Auxiliary Layer for EDAs”.

We have identified three main classes of components that EDAs might wish to consult. The first type is **distance calculation** components. EDAs can use the semantic distance between T-H pairs as their primary decision factor, or can use them as features in more general 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 4.6, “Interface of distance calculation components”.

The second and third component classes 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 4.7, “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 4.8, “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 expressive enough to describe the relevant data structures, and can be used both in-memory and serialized. 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

```
public interface EDABasic<T extends TEDecision>
```

The common interface of EDA is defined through two Java interfaces. `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` as input type, and `TEDecision` as output type. The interface as Java code is listed in Section C.1, “`interface EDABasic` and related objects”.

4.2.1.1. method `initialize`

```
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.

The recommended meta data checking policy for the EDA and the core components is described in more detail in Section 4.10.1, “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. method `process`

```
public T 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 (via `initialize()`). 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 extends `TEDecision` interface which essentially holds the decision as enum value, numeric confidence value and 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:

- `T extends TEDecision`: An object that extends `TEDecision` interface. See Section 4.2.1.5, "interface `TEDecision` ".

4.2.1.3. method `shutdown`

```
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. All resources allocated by the EDA should be released when this method is called.

4.2.1.4. method `startTraining`

```
public void startTraining(CommonConfig c)
```

`startTraining` interface is the common interface for EDA training. The interface signals the start of the training with the given configuration `c`. Note that unlike `process()` method, `startTraining()` method does not need previous call of `initialize()`: start training is independent to `initialize()`.

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 holds single entailment `Pair`.
- (Optional: Multiple T-H cases) It may support the capability to train on a collection of serialized CAS files, where each CAS holds multiple entailment `Pair` annotations.
- 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()`.

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

4.2.1.5. interface `TEDecision`

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

Methods:

- `DecisionLabel getDecision()`: this method returns an object of 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. Note that confidence is about the decision reported in this decision object, and it is not the probability of entailment. For example, high confidence with `CONTRADICTION` decision means that EDA is cofident that the given input is a contradiction.
- `String getPairID()`: this method returns the `entailment.Pair` id as described in the CAS.

Note that the interface only holds the minimal information that is common for all EDAs. EDA implementers can extend this interface to hold additional information for their EDAs. For example, transformation-based EDAs might add transformation steps, and distance-based EDAs might add more detailed information about distance calculations, etc.

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 the method `is()` is listed in Section C.1.3, “enum `DecisionLabel`” The current specification (V1.1) defines 6 labels: `Entailment`, `NonEntailment`, `Contradiction`, `Paraphrase`, `Unknown`, and `Abstain`. These labels form a hierarchy where Entailment and Non-entailment are the top concepts and have several subconcepts. For example, `Paraphrase` is an `Entailment`. This means that `DecisionLabel.Paraphrase.is(DecisionLabel.Entailment)` will return true. Likewise, `Contradiction` and `Unknown` are `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. Note that the `DecisionLabel` is not used in the lexical knowledge resources. It is only used on describing entailment decisions (output of EDAs).

4.2.1.7. Room for improvements: common training and common models

For the moment, we have a very simple `start_training()` interface within the specification. It only signals the EDA to initiate its training procedure. It does not specify where training data is, and where the output should fall into, etc. Such details are assumed to be described in the configuration of the specific EDAs. Neither do we have any common trained model file, or model format. This can be problematic in the long term. Configuration values and model metadata can get fragmented among the EDA implementations, and this will confuse the platform users.

It is therefore desirable to specify common training methods, with a common model format. With common model, the trained model "content" can be different among EDAs, but its header (or meta-data wrappers) should contain common information like, "on what configuration this model was trained", "who is the generating EDA of this model", etc.

This issue has been raised and discussed among WP3 members. The members all agreed that this common training and model feature is what we will pursue. However, it also emerged that it is very hard to realize the common training without some common code base (like common machine training components). For example, currently some of our EDA stores instances with full meta-data as "model" (instead of classifier output model), while some other stores external tools model output (like Weka, or libsvm) as a "model", without wrapper meta-data, etc. The WP3 members have agreed that we will postpone the issue of "common training and model" to later versions of the specification, after delivery of the platform version 0.2 (the first "non-toy" version).

4.2.1.8. Additional interfaces for EDAs

`EDABasic` is the mandatory interface for all EDAs. EDA implementers can choose to provide additional capabilities with optional interfaces. The following is the list of the interfaces currently specified in EXCITEMENT:

- `EDAMultiT`, `EDAMultiH`, `EDAMultiTH`: When an EDA support multiple text and/or hypothesis processing, it should implement the capability with one of the `EDAMulti` interfaces. See Section 4.2.2, “EDA multiple text/hypothesis interface: interface `EDAMultiT`, `EDAMultiH`, `EDAMultiTH` ”.

- `EDAConcurrentProcessing`: An EDA can choose to implement this interface directly for concurrent processing of multiple entailment problems. See Section 4.9.3, “`interface EDAConcurrentProcessing` ”.

- `Reconfigurable`: If an EDA support online reconfiguration (change of configurations within its lifespan), it should provide the capability with this interface. See Section 4.10.2, “Interface `Reconfigurable`”.

There are also auxiliary layer classes and interfaces that provide additional capabilities for user layer code by wrapping EDA interfaces. They are not part of the EDAs themselves, but provide useful methods for complementing and deploying EDAs.

- `InitializationHelper`: Initialization helper enables users to select different EDAs by changing configurations (instead of code changes). see Section 4.3.1, “Initialization Helper”.

- `MultipleTHModeHelper`: This class uses a single pair processing EDA (that only supports EDABasic) and provides methods of `EDAMultiT` and `EDAMultiH`. See Section 4.3.2, “Processing multiple T-H problems with EDABasic:`MultipleTHModeHelper`”.

- `SinglePairProcessHelper`: EDA implementers can implement this interface that provides an easy-to-use access point to their EDA. The interface provides online raw-text input for the EDA, via LAP. See Section 4.3.3, “Produce entailment decision for raw-text input: `SinglePairProcessHelper`”.

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

```
public interface EDAMultiT<T extends TEDecision>
```

```
public interface EDAMultiH<T extends TEDecision>
```

```
public interface EDAMultiTH<T extends TEDecision>
```

Each of the interface defines a method, namely `processMultiT`, `processMultiH`, and `processMultiTH`. EDAs may choose to support them or not, since supporting multiple T/H interfaces are optional. Multiple Texts and Hypotheses are marked in the CAS by multiple `EXCITEMENT.entailment.Pair` annotations (compare Figure 5, “Example of an entailment problem with multiple pairs”).

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

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

```
public List<T> processMultiH(JCas aCas)
public List<T> 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 TEDecision (objects that extends TEDecision). Each `TEDecision` object has its own pair ID (`getPairID()`) --- as annotated in `entailment.Pair` data. Just like the single T-H `process` interface, `processMulti*` should check the input CAS for completeness and raise proper exceptions if some needed CAS structure is missing.

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

Parameters:

- `aCas`: A JCas object with `TETextITem` and/or `TeHypothesisItem` annotations.

Returns:

- `List<T>`: A list of objects with type T where T extends `TEDecision`.

4.3. Auxiliary Layer for EDAs

This section covers auxiliary layer classes and interfaces. They are not part of the EDA interfaces, but provide additional methods that helps accessing EDA functions.

4.3.1. Initialization Helper

```
public abstract class InitializationHelper<T extends TEDecision>
```

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 5.1.6, “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. `InitializationHelper` 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.

`InitializationHelper` solves this problem by providing a common initialization capability. The helper code accepts a configuration file, 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.

4.3.1.1. Methods of Initialization Helper

```
public EDABasic<T> startEngineBasic(File f)
```

```
public EDAMultiT<T> startEngineMultiT(File f)
```

```
public EDAMultiH<T> startEngineMultiH(File f)
```

```
public EDAMultiTH<T> startEngineMultiTH(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 initialization helper, user code can simply open a valid configuration file, and call the initialize helper to get a prepared EDA. Once the `startEngineXX` running 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.

Also note that InitializeHelper is still open for improvements. For example, it might be more realistic to support only startEngineBasic, considering that it is the only common interface that every EDA does support. Similar consideration can be done for generic parameter T: It is likely that startEngine will only support the common TEDecision, since It The type every EDA will share. We leave these possibilities open for now. Once we have a set of first EDAs, we will decide on the common "engine starter". In this spirit, work on the WP4 implementation will shape the final form for initializationHelper.

4.3.2. Processing multiple T-H problems with EDABasic:MultipleTHModeHelper

```
public abstract class MultipleTHModeHelper<T extends TEDecision>
```

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. The interface code is listed in Section C.4, “class MultipleTHModeHelper” It is expected that the platform will share one implementation of mode helper for all EDAs.

4.3.2.1. Methods

4.3.2.1.1. method setEDA

public void setEDA(EDABasic<T> 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().

4.3.2.1.2. Method processMultiT, processMultiH, and processMultiTH

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

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

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

Mode helper implements the EDAMulti* methods. 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.

4.3.3. Produce entailment decision for raw-text input: SinglePairProcessHelper

```
public interface SinglePairProcessHelper<T extends TEDecision>
```

This interface is a convenience interface for "end users" of an entailment engine. The implementation of this interface uses a specific implementation of EDA (EDABasic), and provides a method that allows the user to run the complete engine including LAP and Entailment Core with a single call. The access method receives two String objects as input (the text and the hypothesis, respectively), and returns the result (an object that extends TEDecision) for them.

Since this is a convenience interface, it is limited with regard to functionality and efficiency. It does not support context sentences, or multiple text/hypothesis situations. Also, calling the LAP for just one sentence pair is presumably quite inefficient.

Note that, unlike MultipleTHModeHelper, the platform cannot share a single ProcessHelper implementation for all EDAs. Each EDA needs to run a different LAP, thus each EDA is expected to have its own ProcessHelper.

4.3.3.1. method processRawInput

```
public T processRawInput(String text, String hypothesis)
```

The method first calls needed LAP pipeline to analyze the given raw text and hypothesis. Then, it uses EDABasic process() method to deliver the result (TEDecision or its extension).

Parameters:

- text: a java String that holds the text.

- hypothesis: a java String that holds the hypothesis.

Returns:

- T: An object of the interface that extends TEDecision.

4.4. Common functionality of components: The Component interface

There is a small set of methods that are common to all components. These methods are primarily concerned with the administrative aspects of components and their interactions with the EDA. These methods form the Component interface. All more specific interfaces in the following sections are subinterfaces of interface Component.

4.4.1. method getComponentName

```
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 4.10.3, “Component name and instance name”.

4.4.2. method getInstanceName

```
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 4.10.3, “Component name and instance name”. Note that this method can return null value, if and only if all instances of the component shares the same configuration.

4.4.3. Initialization of a Component

Previously, the method void initialize(CommonConfig c) was included in the interface Component. This means that all components are supposed to be initialized by this method, with a configuration object. Since specification verion 1.1.2, the method is removed from the interface. This changes the expected behavior of a component for its initialization.

All component implementations should satisfy the following conditions.

1. All configurable/settable parameters are exposed in the constructor. So basically, a user can initialize and run a component; even without a configuration (`CommonConfiguration`) object, as long as the user supplies correct values for the arguments.

2. Components that need to read from the configuration *shall* provide a overridden constructor, which gets a `CommonConfig` object. (For example, `GermaNet(CommonConfig c)`, or `EnglishWordNet(CommonConfig c)`.) This constructor code knows which part of the common configuration is relevant to the component, and reads the configuration values from the configuration, and call the first constructor.

Note that the overridden constructor that knows how to deal a the configuration object can have forms other than the single argument of `CommonConfig`. For example, components that permits multiple instances might need additional arguments. It might look like `GermaNet(CommonConfig c, String instanceName)`, where the `instanceName` will identify proper configuration subsection for that instance, etc.

Also note that EDA still keeps its `initialize()` method, in the `EDABasic` interface. This will force all EDA implementations to have this method. This reflects that EDA will always have a dependency to the configurations, and the method will be used by top-level initialization helper and other user-layer codes.

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.

4.5. Interface for Scoring Components

This component accepts a T-H pair (represented in a `JCas`) as the input, and returns a vector of scores (as `Vector<Double>`). The scores can be anything that represent some aspects of the given T-H pair. We can easily imagine various scores that report possible interesting "features" of the given pair. For example, number of shared arguments and predicates, location of main predicate, length (or length difference) of T/H, etc.

In a sense, you can regard this type of component as a "feature extraction component" where the extracted features are reported back as a vector of numbers. What each number of the resulting vector means is different among components. This, should be documented by each component implementers in JavaDoc of the component.

One special case for reporting back scores, is the distance calculation (or similarity calculation) between a T-H pair. Since distance calculation between T-H is such a common capability needed for various entailment methods, we define it as a separate interface that extends the scoring component interface. See `interface DistanceCalculation` (Section 4.6, “Interface of distance calculation components”) for that interface.

4.5.1. `interface ScoringComponent`

Like all other components, the interface extends `interface Component`. The interface adds a single method `calculateScores()`, which gets a `JCas` as the input argument, and returns a `Vector<Double>`.

4.5.1.1. Method calculateScores()

```
public Vector<Double> calculateScores(JCas cas)
```

The method delivers a list of numbers where the numbers represent some features of the given JCas object. The calculation can be only done if the CAS has two prepared views (`TextView` and `HypothesisView`), and the needed linguistic annotations. Note that the method compares all contents of the two views directly, and does not (and should not) utilizes other entailment annotations like `entailment.Pair`, or `entailment.Text`, etc. This is to retain the generality of the scoring component and the distance components (so that they can be used in comparing any two given texts).

The method may choose to check the validity of the input. For example, if the provided annotations are not compatible for the calculation (e.g. if the parse tree is missing for a tree level feature extractor, etc), it can raise an exception accordingly.

Multi-pair CASes should not be passed directly to this method, since it only extracts scores from the two views. (It is expected that LAP code will provide a tool that breaks down a multi-pair CAS into an iterative set of single-pair CASes. One can use such tool for multi-pair CASes).

- parameter `cas`: a JCas with `TextView` and `HypothesisView`. The texts from the two views will be used in the score calculation
- returns `Vector<Double>`: a vector of double values. They can serve as feature values that represent the given pair. Each implementation should documentate meaning (or underlying calculation) of the values.

4.6. Interface of distance calculation components

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

4.6.1. interface `DistanceCalculation`

The interface is a subinterface of `ScoringComponent`. 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.6, “`interface DistanceCalculation` and related objects”.

4.6.1.1. method `calculation()`

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

The method delivers the distance calculation result in an Object `DistanceValue` (Section 4.6.2, “Type `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. The `calculation()` method knows nothing about other entailment annotations (like `entailment.Pair` or `entailment.Text`) and it should not check those annotations.

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

4.6.1.2. Inherited method `calculateScores()`

The main data structure that is returned from a distance component is `DistanceValue`, of method `calculation()`. However, an implementation should report back a vector correctly when it is called by the inherited method `calculateScores()` (Section 4.5.1.1, “Method calculateScores() ”). The choice of values to report back is, of course, the implementer's choice. However, it is recommended that the returned set of vectors should have the distance values (duplicated to the ones in DistanceValue), and all underlying raw scores (if any) that helped calculating the DistanceValue. (For example, unnormalized values, raw-frequency count, length of T/H, denominator of normalizing, etc. They may be used as additional features by some EDAs).

4.6.2. Type `DistanceValue`

This type holds the distance calculation result. It has some member variables and public access functions for the variable. The type is used to exchange data between the component and the EDA. Its variables are set during initialization. The actual java code listing is included in Section C.6, “`interface DistanceCalculation` and related objects”.

4.6.2.1. Variables and access methods

- `private double distance`: The normalized distance. The maximum value is 1 (maximally different), and the 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()`.

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 the boolean variable `isSimBased`. For example, the `unnormalizedValue` of cosine similarity will presumably hold the original vector product value, with `isSimBased` as `true`. In this case, a higher `unnormalizedValue` means more similar (less distant) objects.

4.7. 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 the `class LexicalRule`, and collections of such rules are represented by the interface `LexicalResource`.

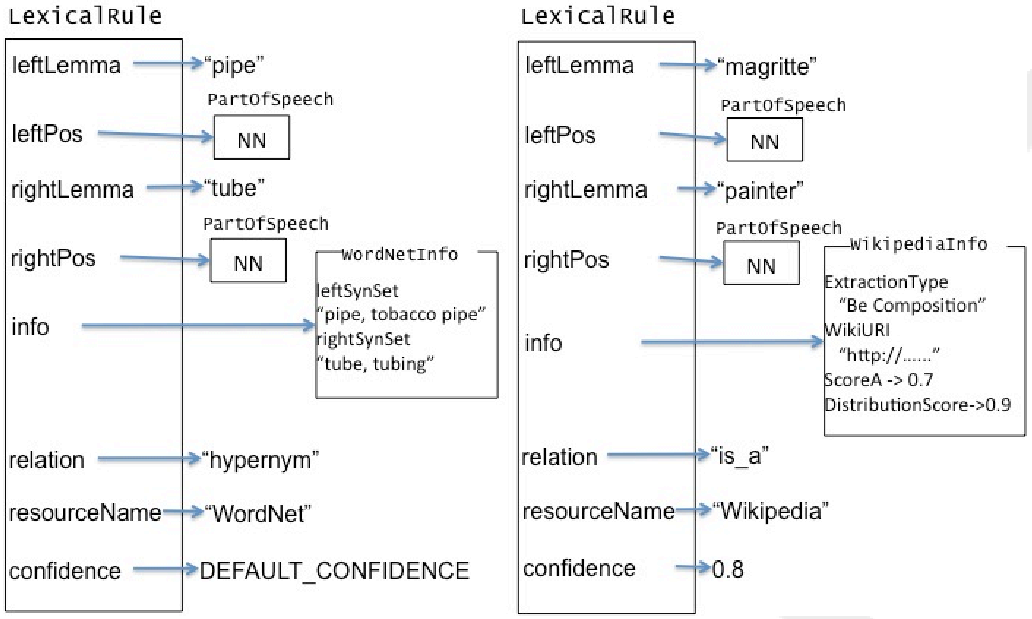
4.7.1. Type `LexicalRule`

```
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 (LHS) and a right hand side (RHS). The basic arrangement of lexical resource is that the left hand side has a relationship to the right hand side. Normally, this relationship is entailment (LHS entailment RHS; return values of Section 4.7.2, “Interface `LexicalResource`”). However, they can also represent resource specific or canonical relations (see Section 4.7.3, “interface `LexicalResourceWithRelation` ” for querying such relations). A lexical rule also provides additional pieces of information include confidence, relation name, and resource name. Finally, lexical rules are parametrized by a type `I` (short for information) which allows the type to hold additional resource-specific properties.

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

Figure 8. Examples of lexical rule instance



On the left, it shows an instance of `LexicalRule` that represents the relationship between "pipe" and "tube". The value `originalRelation` holds the relation (here the "hypernym") from left ("pipe") to the right ("tube"), as recorded in the knowledge resource (here, WordNet). In this example, the left hand side entails the right hand side. The confidence is not provided by the implementation, and it is given as `DEFAULT`. Additional information is provided in the example with info variable. The type variable `I` is instantiated by a type "WordNetInfo" which holds information about synsets.

On the right, another example shows that "Magritte" entails "painter". In this case, the knowledge is captured from Wikipedia, and the resource dependent information provides various Wikipedia-based links and scores. 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.7, "`class LexicalResource` and related objects".

4.7.1.1. 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. This `I` must be an extension of `RuleInfo`.
- `private final String relation`: 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 double DEFAULT_CONFIDENCE`: A class constance 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]. Consider this value as a probability of entailment from LHS to RHS. Naturally, higher value

means more confident on the entailment of LHS to RHS. If no meaningful confidence score is provided by the resource, this 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.

4.7.1.2. 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 this canonical set of POS labels.

4.7.2. 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.7, “`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)`: Returns 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>> getRulesForRight(String lemma, PartOfSpeech pos)`: Returns 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>> 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.

- `void close()`: this method enables the implementation to gracefully close underlying system resources (files, database connections, etc). Users of syntactic and lexical resources are expected to call close() when the resource is no longer needed.

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

Note that all lexical rules returned by this interface represent entailments (LHS->RHS entailments). In this sense, the methods only support querying of "entailments". If your lexical resource supports additional relational queries, you can provide them with the additional interface Section 4.7.3, “interface `LexicalResourceWithRelation` ”.

4.7.3. interface `LexicalResourceWithRelation`

`public interface LexicalResourceWithRelation<I extends RuleInfo, R extends RelationSpecifier> extends LexicalResource<I>`

The interface LexicalResource (Section 4.7.2, “Interface `LexicalResource`”) defines the common method of querying and getting lexical entailment rules of the given words. This abstraction is useful in the sense that it abstracts various underlying lexical resources with the entailment relation. This enables the caller of LexicalResource implementations to query various knowledge resources in the same fashion.

However, this comes with a price: the LexicalResource interface cannot provide querying capability of relations others than entailment. For example, user of that interface cannot query NonEntailment words of the given term (terms that you are sure that is not entailment), nor resource-specific relations (like synonym or hypernym of WordNet, or Stronger-than or happens-before of VerbOcean, etc). These resource-specific relations are reported back as part of the query result (RuleInfo within LexicalRule), but cannot be directly asked for.

`LexicalResourceWithRelation` is defined to recover this capability. It permits lexical resource implementers to define additional query methods with relations. It enables not only the canonical relation queries but also queries with resource-specific relations.The interface extends `LexicalResource`, and adds three methods. Essentially, each of the method has one additional argument that represents the relation to be queried.

```
public interface LexicalResourceWithRelation<I extends RuleInfo, R extends RelationSpecifier> extends LexicalResource<I>
```

- `List<LexicalRule<? extends I>> getRulesForLeft(String lemma, PartOfSpeech pos, R relation) throws LexicalResourceException;`
- `List<LexicalRule<? extends I>> getRulesForRight(String lemma, PartOfSpeech pos, R relation) throws LexicalResourceException;`
- `List<LexicalRule<? extends I>> getRules(String leftLemma, PartOfSpeech leftPos, String rightLemma, PartOfSpeech rightPos, R relation) throws LexicalResourceException;`

Each of the method gets one additional parameter `R relation` that specifies the relation to be fetched. For example, if `relation` is "NonEntailment", the resource will return rules that specifies "NonEntailment" (where LHS->RHS is confidently cannot be entailment). If the `relation` holds "Synonym", it means the query should return Lexical rules where LHS to RHS is synonymy.

Note that the relation is represented by a generic parameter `R`. Each `R` is an extension of `RelationSpecifier`, where the `R` can be tailored for each resource. We adopt a simple hierarchy for this `R`. The interface `RelationSpecifier` and related objects are described in the following subsections.

4.7.4. Interface RelationSpecifier and its hierarchy

The following figure shows the simple hierarchy of `RelationSpecifier`. The interface `RelationSpecifier` stays in the top, and two interfaces (`CanonicalRelationSpecifier` and `OwnRelationSpecifier<E>`) extends this top interface.

```
                    RelationSpecifier
                     /             \
                    /               \
CanonicalRelationSpecifier     OwnRelationSpecifier<E>
```

The intention of adopting the hierarchy on relation specifier is to explicitly mark the class of relations that are supported by each `LexicalResourceWithRelation`. For example, all lexical resources that support canonical relation bears the same signature of `R` (as `<R extends CanonicalRelation>`). This is the same for resources that support resource-specific relations (`R` as `<R extends OwnRelationSpecifier<E>>`).

Let's check each of the interfaces.

4.7.4.1. interface RelationSpecifier

```
public interface RelationSpecifier
```

This is an empty interface where it defines no methods. Extending or implementing this interface means that the class is to be used as the paramter `R` of `LexicalResourceWithRelation`. This interface is the top of the relation specifier hierarchy, and intentionally left empty. This means that you can design a speical RelationSpecifier that can be used in `LexicalResourceWithRelation` if needed. However, in most cases, you will implement one or both of the following sub-interfaces.

4.7.4.2. CanonicalRelationSpecifier

```
public interface CanonicalRelationSpecifier extends RelationSpecifier
```

This interface defines a single method:

- `public TERuleRelation getCanonicalRelation();`

The method returns the canonical relation (as `enum TERuleRelation`, Section 4.7.4.3, “`enum TERuleRelation`”), where it means this canonical relation is being queried upon. This return value can be `null`, which means "don't care" (the query don't care about canonical relation).

4.7.4.3. `enum TERuleRelation`

This enum value represents a canonical mapping of lexical relations. Lexical rules of different `lexicalResource` may have different original relations. However, they can support this common `TERuleRelation` which describes the relation in terms of textual entailment. It can be one of the two values:

- `Entailment` which means that the lexical relationship from LHS to RHS is entailment.
- `NonEntailment` means that the lexical relationship from LHS to RHS cannot be entailment.

Note that `NonEntailment` means the knowledge resource is confident that the relationship is not entailment (e.g. contradiction, etc). Also note that this enum only describes the lexical relation, not entailment decision. For entailment decisions, there is a separate enum `DecisionLabel` (Section 4.2.1.6, “enum `DecisionLabel`”).

4.7.4.4. `interface OwnRelationSpecifier`

```
public interface OwnRelationSpecifier<E> extends RelationSpecifier
```

The interface defines a single method:

- `public E getOwnRelation();`

The method returns the resource-specific relation (as `E` which is defined by a generic parameter). The returned `E` will be the resource specific relation that is being queried upon. This return value can be `null`, which also means "don't care".

The paramter `E` is to be defined by each resource implementer to represent resource specific relations.

4.7.4.5. Conformance on LexicalResource and LexicalResourceWithRelation

- All lexical resources must support the minimal interface `LexicalResource`. This interface only returns lexical rules of LHS->RHS entailment relation.
- If the lexical resource supports canonical relation (TERuleRelation), it should implement `LexicalResourceWithRelation` with a relation specifier that implements `CanonicalRelationSpeicifier`.
- If the lexical resource supports resource-specific queries, it should implement `LexicalResourcewithRelation` with a relation specifier that implements `OwnRelationSpecifier`.
- If the lexical resource supports both canonical and resource-specific queries, its relation specifier should implement both `OwnRelationSpecifier` and `CanonicalRelationSpecifier`. Note

that if both of the relation are specified, the query should be interpreted with "AND", which means both of the relation constraints should be satisfied. (For example, Entailment AND Synonym, or NonEntailment AND meronym, etc)

The following is a small code example that shows an implementation of `LexicalResourceWithRelation`, which supports both canonical relation and resource specific relation.

4.7.4.6. An Example of `LexicalResourceWithRelation` and `RelationSpecifier`

Let's see an example for a clearer explanation. The following is a partial (and artificial) implementation example of `LexicalResourceWithRelation` where its `R` supports both `CanonicalRelationSpecifier`, and `OwnRelationSpecifier`.

The following is `RelationSpecifier` of this resource (German WordNet). Note that `WordNetRelation` is an enum that has all WordNet relations, and used as <E> of `OwnRelationSpecifier<E>`:

```
public class GermaNetRelation implements CanonicalRelationSpecifier,
                         OwnRelationSpecifier<WordNetRelation>
{

   public GermaNetRelation(TERuleRelation canonicalRelation,
                           WordNetRelation ownRelation)
   {
      this.canonicalRel = canonicalRelation;
      this.ownRelation = ownRelation;
   }

   public GermaNetRelation(TERuleRelation canonicalRelation) {
      this(canonicalRelation, null);
   }

   public GermaNetRelation(WordNetRelation ownRelation) {
      this(null, ownRelation);
   }

   @Override // from canonical relation specifier
   public TERuleRelation getCanonicalRelation() {
      return canonicalRel;
   }

   @Override // from own relation specifier
   public WordNetRelation getOwnRelation() {
      return ownRelation;
   }

   private final TERuleRelation canonicalRel;
   private final WordNetRelation ownRelation;
}
```

GermaNet implementation would be defined with the following generic parameters:

```
public class GermaNetImplementation implements
   LexicalResourceWithRelation<GermaNetInfo, GermaNetRelation> { ...
```

And the usage example would be like the followings:

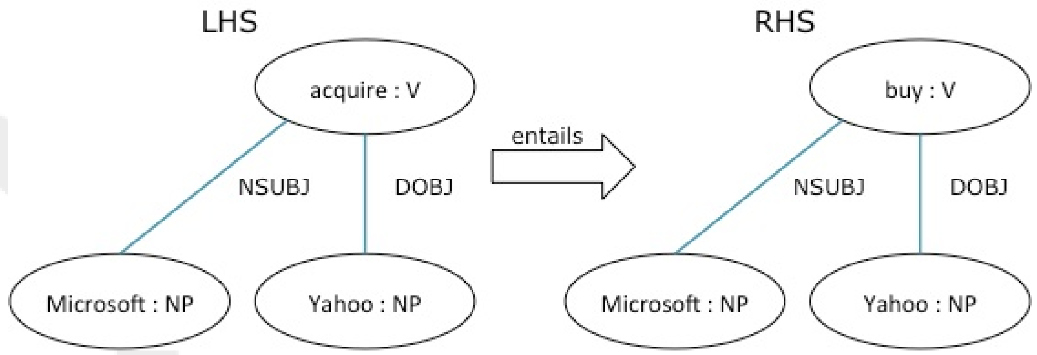
```
GermaNetImplementation germaNet = new GermaNetImplementation();

String lemma = "beautiful";
PartOfSpeech pos = new GermanPOS("adjective");

// query for synonym of "Beautiful"
GermaNetRelation rel1 = new GermaNetRelation(WordNetRelation.SYNONYM);
germaNet.getRulesForLeft(lemma, pos, rel1);

// query for NONENTAILMENT terms to "Beautiful"
GermaNetRelation rel2 = new GermaNetRelation(TERuleRelation.NONENTAILMENT);
germaNet.getRulesForLeft(lemma, pos, rel2);

String lemma = "apple";
PartOfSpeech pos = new GermanPOS("noun");

// query for ENTAILMENT and hypernym of "apple"
GermaNetRelation rel3 = new GermaNetRelation(TERuleRelation.ENTAILMENT,
                                             WordNetRelation.HYPERNYM);
germaNet.getRulesForLeft(lemma, pos, rel3);
```

4.8. Interface of syntactic knowledge components

Some entailment relationships cannot be described well at the lexical level, but require access to the syntactic level. Examples include changes of verb voice, changes in a predicate's argument structure, or proper paraphrase.

In the EXCITEMENT platform, we call syntactic level knowledge *syntactic rules*, and a collection of such rules with standard access methods a *syntactic rulebase*. Similar to lexical rules, syntactic rules also have two sides (LHS and RHS) and define the relationship between the them (for example, LHS entails RHS, or LHS does not entail RHS). Unlike in the case of lexical knowledge, each side (LHS / RHS) is defined as a partial parse tree. The figure Figure 9, “Example of a syntactic rule” shows a simple example.

Figure 9. Example of a syntactic rule

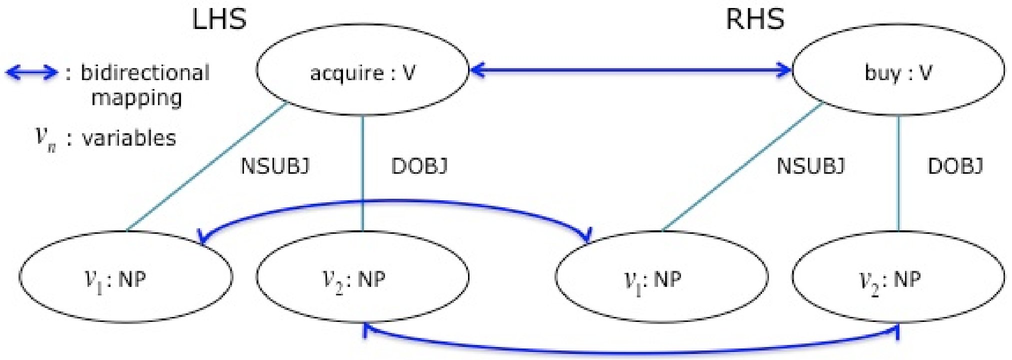


In the figure, two partial parse trees represent a simple syntactic rule. The rule is about replacing "Microsoft acquires Yahoo" to "Microsoft buys Yahoo". Note that both LHS and RHS are represented as partial parse trees, which need to be matched with concrete input parse tree. For example, the LHS will match the following sentence; "Following the decisions of the new CEOs, Microsoft will acquire Yahoo soon.", since the parse tree of the sentence has the partial parse tree of the LHS.

Application of a syntactic rule takes place by replacing the matched partial parse tree of the sentence to that of RHS. The rule described in the figure will replace the verb with two arguments, and declares that the rewritten tree has entailment relationship with original tree. One can visualize a syntactic rule as a parse tree rewriting rule with entailment relationship declaration.

A syntactic rule also has two additional fields that does not exist in a lexical rule: variables and mappings. The following figure (Figure 10, “Variables and mappings of an syntactic rule instance”) demonstrates them.

Figure 10. Variables and mappings of an syntactic rule instance



A variable is a parse tree node that can match any dependency (sub)tree whose root node matches its part of speech constraint. For example, the LHS of the figure contains two variables. It matches any dependency tree that contains an instance of the verb 'acquire' whose subject and direct object are both proper nouns (POS: NP). Note that these proper nouns can have arbitrary modifiers. Clearly, the introduction of variables is crucial for the ability to represent generalizable lexical-syntactic rules.

Correspondences between LHS and RHS nodes are established through a bidirectional mapping. It specifies how the variable nodes of the source tree will be copied into the target tree. The semantics of the rule are that the complete dependency trees rooted in variable nodes must be copied to the corresponding variable nodes in the target tree. For example, consider the sentence "Software giant Microsoft acquired Yahoo". If only the variable nodes themselves were copied, the result would be "Microsoft bought Yahoo".

4.8.1. Type `SyntacticRule`

`public class SyntacticRule`

The class represents an instance of a syntactic rule.

4.8.1.1. Variables and access methods

- `BasicNode leftHandSide`: LHS of this rule. A parse tree represented with `BasicNode` objects. The value is set by the constructor, and can be read by `getLeftHandSide()`.
- `BasicNode rightHandSide`: RHS of this rule. A parse tree represented on the basis of `BasicNode` objects.
- `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 possible, the object should be immutable (all variables as private final) so that values can be only set by the constructors.

4.8.1.2. Related Objects

- `BidirectionalMap<K,V>`: A map similar to normal hash map, but it also has back-links to model a bidirectional mapping. The specification does not specify the implementation. Existing code is available from Apache commons, Google, and the BIUTEE infrastructure.

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

Note that the class `SyntacticRule` is described here without the generic parameters. But it must be a generic class that is parameterized by `BasicNode` and its related objects. The generics are omitted here for the readability.

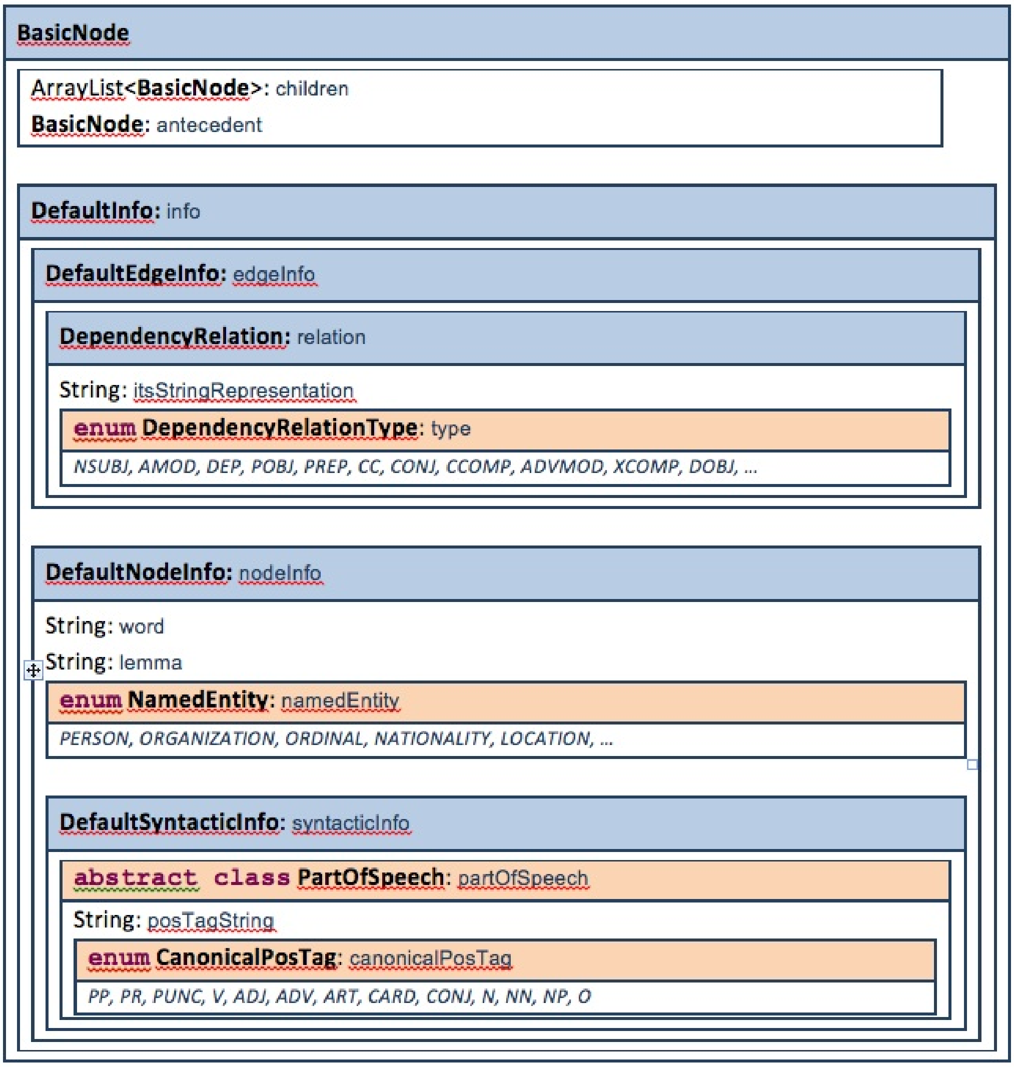
4.8.1.3. Basic Node: Common representation for dependency parse tree

To share a common representation for syntactic knowledge, a shared syntactic representation is required. 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.

The term "basic" hear means that it contains only the basic information - information that is common to almost all languages. It does not contain information that appear only on some languages like grammatical role, morphological form, etc. It is expected that specific language nodes will be implemented by extending the `BasicNode`.

Figure 11, “Contents of a basic node” gives a graphical overview of the basic node.

Figure 11. Contents of a basic node


BasicNode
ArrayList<BasicNode>: children
BasicNode: antecedent
DefaultInfo: info
DefaultEdgeInfo: edgeInfo
DependencyRelation: relation
String: itsStringRepresentation
enum DependencyRelationType: type
NSUBJ, AMOD, DEP, POBJ, PREP, CC, CONJ, CCOMP, ADVMOD, XCOMP, DOBJ, ...
DefaultNodeInfo: nodeInfo
String: word
String: lemma
enum NamedEntity: namedEntity
PERSON, ORGANIZATION, ORDINAL, NATIONALITY, LOCATION, ...
DefaultSyntacticInfo: syntacticInfo
abstract class PartOfSpeech: partOfSpeech
String: posTagString
enum CanonicalPosTag: canonicalPosTag
PP, PR, PUNC, V, ADJ, ADV, ART, CARD, CONJ, N, NN, NP, O

A node basically holds two sets 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.

- `antecedent`: this member variable holds a reference to "duplicated node" in the dependency tree. For example, in a sentence like "He managed to go", it is possible to generate a duplicated node "He". The antecedent field of the duplicated one will point the original node.

- `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 holds the an enum value that represents dependency relationship. The enum `DependencyRelationType` closely reflects the adopted UIMA dependency types (the subtypes of `type.dependency.Dependency`).

The `nodeInfo` holds the following information:

- `word`: this string holds the word (token) itself.

- `lemma`: the lemma of this token

- `namedEntity`: it holds a enum value that represents a NER class. If this node is not a named entity, this value is null. The enum type `NamedEntity` closely reflects the adopted UIMA NER types (subtypes of `ner.type.NamedEntity`).

- `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 (not shown in the figure), `nodeInfo` also provides two methods `isVariable()` and `getVariableID()`. `isVariable()` method returns true if and only if the node represents a variable of a syntactic rule. If the node is a variable, `getVariableID()` returns ID of the variable as an integer. (The ID is only for human readers and rule visualizations -- actual rule matches and applications are done by bidirectional mappings provided in the rule.)

The `BasicNode` is implemented with two generic parameters. One is the info type (that holds nodeinfo and edgeinfo), and the other is node type itself (children and antecedent). Generic parameters are omitted in the text paragraph for readability, but they are described in the appendix source code in Section C.8.2, “class `BasicNode` and related classes”.

4.8.2. 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.

4.8.2.1. 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.

- `List<RuleMatch> findMatches(BasicNode textTree, BasicNode hypothesisTree)`: This overloaded version of `findMatches` method gets two trees instead of one. The two trees are text and hypothesis tree, and the method tries to find matches such that LHS matches the text tree and RHS matches the hypothesis tree.

- `void close()`: this method enables the implementation to gracefully close underlying system resources (files, database connections, etc). Users of syntactic and lexical resources are expected to call close() when the resource is no longer needed.

The overloaded second method is provided for efficiency. Since it might be too expensive to find and apply all possible matches, the overloaded method only returns the rules that also match the hypothesis. This way, the user of the knowledge base can applies rules that directly make the text more similar to the hypothesis. The drawback of this approach is that it misses some rules that can make text more similar to hypothesis. For example, it will miss cases where subsequently applying two or more rules that make text more similar to hypothesis.

4.8.2.2. Related Objects

- `RuleMatch`: this simple object represents a match of a syntactic rule on a parse tree. It has two variables. Member variable `rule` is a `SyntacticRule`, which holds the rewriting rule. The other variable `mapLHStoTree` is a bidirectional mapping of `BasicNode`. The variable holds mapping info between each LHS node of the `rule` and the corresponding node of the input parse tree.

4.9. Concurrent Processing

4.9.1. The issue of concurrent processing

Modern computer hardware is very well prepared for concurrent and parallel processing. Even modest PCs now have multiple cores with multithreading capability. Given users' requirements of quick or even real-time processing, it is important that the EXCITEMENT platform supports concurrent processing.

The EXCITEMENT architecture (as seen in Figure 7, “Common components within EXCITEMENT entailment core.”) can incorporate concurrent processing at two different levels. We will call them concurrent component execution, and concurrent EDA execution. They have different advantages and disadvantages.

Concurrent component execution. Imagine an EDA that requires the results of three sub-components for its entailment decision (a typical situation for TIE or EDITS). Without concurrent processing, the EDA must call three sub-components sequentially and has to postpone its decision to when all three results are available. Concurrent component execution can reduce the total running time by reducing the time needed to produce a single TE decision.

Concurrent EDA execution. Imagine an EDA which must process a set of textual entailment problems. If the time needed to process the set is n, we can reduce the required time to `n/m` by running `m` threads for the EDA. This level of concurrency will reduce the time to produce a **set** of TE decisions by concurrent processing of the selected EDA.

4.9.2. Our approach on concurrent processing

We have decided on the following contract for concurrent processing.

- All components must declare clearly whether they are *thread-safe*: If a module is thread-safe, instances of this module can be called from multiple threads. If it is not thread-safe, each thread needs its own instance. Thus, each component provider must carefully document the thread safety.

We have adopted a Java annotation for this (`@ThreadSafe, @NotThreadSafe`, see Section 7.1.5, “Code annotations”).

- No additional interface for concurrent component execution: We decided that running components concurrently is the responsibility of the EDA. Thus, we do not define additional concurrent interface for components. Once the preparation of multiple threads for concurrent component calling is handled by the EDA code, no additional interface is needed on the component side. Whether an EDA calls a component concurrently or sequentially, it will use the same interface. Having thread-safety correctly marked for the component is sufficient.

  Amount of memory and number of CPU cores would be limiting factor for general concurrent running. To help our users, each EDA should clearly document its memory requirement for each run, also with regards to the input size, and number of concurrent runnings. With this information documented, the users will be able to prepare a good configuration.

- An interface for concurrent EDA execution: Concurrent processing of multiple TE problems needs a new interface. The EDA interfaces defined in Section 4.2, “EDA Interface ” only process a single TE problem. The new interface makes it possible to process a set of TE problems concurrently, returning multiple decisions. The interface is defined in the next subsection.

4.9.3. `interface EDAConcurrentProcessing`

The goal of this interface is to provide an access method for EDAs (and/or EDA wrappers) that can process a set of entailment decision problems concurrently.

```
List<TEDecision> processDataSetConcurrently (List<JCas> casList)
```

The interface receives a set of TE decision problems as a list of JCAS. Each CAS object holds an entailment problem (as defined in Section 3.3.4, “Additional Types for Textual Entailment”). The result is returned as a list of `TEDecision` objects. The interface is not an asynchronous interface: it will block until it process all the given problem list, and return the results only after all processing is done.

The expected behavior of implementation of this interface is to process the given dataset concurrently with a number of threads, using general Java concurrent capabilities like Executors and thread pools. The number of threads and other configurations should be defined in the common configuration. Proper initialization (for example, initializing multiple instances for non thread-safe components) should be ensured by the implementation so that the user does not need to care about sub-components and how they are being run.

This interface can be directly implemented by an EDA (i.e., a class that already implements EDABasic and/or other interfaces). It can be also implemented by a concurrent running wrapper (a "runner") for the EDA, if the implementation of this interface does not lend itself naturally to the internal structure of the EDA.

4.9.4. Future enhancements

The WP3 members have agreed that we will approach this issue with incremental steps. We can imagine a set of future improvements including:

- Asynchronous interfaces
- Generic concurrent "runner" that can run any single-thread EDA of `EDABasic` with multiple EDA instances.
- Different concurrent approaches for multiple text and/or multiple hypothesis cases.

Additional concurrent processing supports will be realized as the platform matures.

4.10. Initialization and metadata check

4.10.1. 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 direct components (components that will be directly called by the EDA). Each component, in turn, will do their own checks, using the information from the common configuration file (or equivalent arguments). Likewise, if a component uses a sub-component, it will also initialize the sub-component.

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.

4.10.2. Interface `Reconfigurable`

Some components may need the ability to reconfigure. For example, a user of a distance calculation component may wants to compute 10 distance values, by using 10 different "weighting schemes" using the same underlying distance calculation component. One way of doing this would be to generate ten instances of the same component. However, this might not be desirable (e.g. if the component does not permit more than a single instance, or if the initialization of the component takes a lot of time).

The interface `Reconfigurable` provides a way to modify the component configuration at runtime. The term "reconfiguration" here means changing some of the configurable parameters of a component, without re-initializing all resources, or changing other settings that may remain constant.

Any entailment core component that supports reconfiguration should implement this capability though this interface. This also includes EDAs. If EDA needs to support online reconfiguration, it should support the capability with this interface. Note that when an EDA provides this interface, the EDA should also update the states of its sub-components according to the configuration method argument. It is the responsibility of EDA implementers to ensure that configuration changes are handed through to sub-components, if applicable. This may mean simply reconfiguring some sub-components (which support this) and re-initializing others (which do not).

4.10.2.1. Methods of Interface Reconfigurable

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

The interface provides a single method, `reconfigure()`. The method gets the same single argument (an instance of `CommonConfiguration`), but the contract is different to first-time initialization. Instead of 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 at the first initialization), 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.

4.10.3. Component name and instance name

The 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`). Note that this string can be null if and only if the component does not permit multiple instances with different configurations.

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.

4.11. Context Sensitive Lexical Resource

4.11.1. Basic requirements

The lexical resource interface (`LexicalResource`) is imported from BIUTEE to EOP, and it is well-tested by both BIUTEE and other EOP EDAs. It is expanded to include "resource-specific" relations (LexicalResourceWithOwnRelation) if a resource can support, and currently the interfaces are quite stable.

However, the interfaces are not providing any means of passing the context from EDA to the resource. Thus, the existing interfaces are not adequate to represent "context-sensitive" lexical resource. So we need to define one extension that permits context-passing, and all related matters.

Let's briefly outline current assumptions and requirements before outlining the actual interface.

- We use the same return type: no change on the result of the queries (list of LexicalRule).
- We make this interface as an "extension" interface. A resource still needs to provide basic LexicalResource interface. The new interface adds an additional set of methods that can behave "context-sensitive" mode.

The followings are requirements, rather than assumptions.

- Form of context should not be limited: it can be anything. Thus, passing the T-H pair itself and all its annotations would be ideal.

- Capability to pass "candidates" are highly desirable: context sensitive resource is not a static resource, and requires dynamic calculation. Thus, "limitation" of rule calculation target is very desirable for its efficiency. This can be done by passing a set of candidates that limits RHS or LHS. (e.g. limit RHS or LHS with this set of terms)

- "position-only" query is also quite desirable. (I want to get a set of rules, if any, between term$_a$ in location$_a$ and location$_b$. I don't care about what is the current term in location$_b$.)

4.11.2. Interface outline

The following interface outlines an interface that satisfies all of the above assumptions and requirements.

4.11.2.1. `Interface LexicalResourceWithContext extends LexicalResource`

The interface adds three methods to the usual `getRules` family. Each added method (`getRulesLeft`, `getRulesRight` and `getRules`) overrides those methods in the basic interface, with added arguments that passes the context of the term in interest. Among the three added methods, `getRules` is the most general (and powerful) one that can pass two contexts (both of Text and Hypothesis), while `getRulesForLeft` and `getRulesForRight` are dealing with one side.

4.11.2.1.1. method `getRules` family added by `LexicalResourceWithContext`

List<LexicalRule<? extends I>> getRules(String leftLemma, PartOfSpeech leftPos, String rightLemma, PartOfSpeech rightPos, JCas context, int lemmaLBegin, int lemmaRBegin, List<Tuple<String,PartOfSpeech>> candidates) throws LexicalResourceException;

Let's see the expected usage of the method with an example. The T-H pair is given as the followings.

```
___01234567890123456789012345678901  -- CAS SOFA position
T: John walked his dog in the park.
H: John took hit pet for a walk.
```

Let's assume that a caller of the resource is interested in "dog" - "pet" part of the T-H pair. The following examples all assumes this case, with the above T-H pair.

Example #1: `getRules("dog", noun, "pet", noun, aJCas, 16, 14, null);`

- this call specifically asks the resource to check relationship between (dog, N) -> (pet, N); in the context of the annotated T-H pair (JCas aJCas); position 16 of Text -> position 14 of Hypothesis.

- (Note that candidate list is null, which basically means consider all candidates --- however, since lemma of both sides is given, list of candidates are not meaningful in this case. The resource will only consider dog -> pet, within the context).

Example #2: `getRules("dog", noun, null, null, aJCas, 16, 14, candidates);` where `candidates` are: `[ ("pet", noun), ("animal", noun),... (lemma_n, pos_n) ]`

- This call asks the resource to check relation "dog" -> candidates (which holds pet, animal, etc); in the context of annotated T-H pair aJCas; position 16 of T -> position 14 of H.

- Note that, candidate is non-null; which dictates that the resource will consider only those candidates as possible "filler" of right lemma/pos (which are null).

- If the candidates variable is "null"; the resource will try to look for all rules "dog -> (any word)": this might be a complex operation and the caller should expect slower process, compared to the candidate given call case.

Example #3: `getRules(null, noun, null, noun, aJCas, 16, 14, candidates);` where `candidates` are: `[ ("pet", noun), ("animal", noun),... (lemma_n, pos_n) ]`

- This call asks the resource to check possible rules that can be used on the context of position 16 of text -> position 14 of hypothesis, as nouns.
- Note that both lemma are null, but the candidates are given. Thus, it will use all of the candidates, as possible filler for those null lemmas.

Example #4: `getRules(null, null, null, null, aJCas, 16, 14, candidates);` where `candidates` are: `[ ("pet", noun), ("animal", noun), ... (lemma_n, pos_n) ]`

- An extreme call case, where only the context is given (as the TH-pair and position) with a set of candidates.
- The resource will try to find rules that can fill (left -> right), where left location and right location is known.
- The candidates are given in this case -- thus the resource will only regard those candidates as the possible 'fillers' for the positions.

4.11.2.1.2. `getRulesLeft` family added by `LexicalResourceWithContext`

- `List<LexicalRule<? extends I>> getRulesForLeft(String lemma,PartOfSpeech pos, JCas context, int lemmaLBegin,List<Tuple<String,PartOfSpeech>> candidates) throws LexicalResourceException;`

Let's see some more examples for this case. Note that the methods does **not** care about Hypothesis side, and it only sees Text side. Thus, the meaning of a call is "what are the lexical rules that replace the term in its own context?".

```
___01234567890123456789012345678901  -- CAS SOFA position
T: John walked his dog in the park.
H: [ don't care ]
```

Example #1. `getRulesForLeft("dog", noun, aJCas, 16, null)`

- get me all rules that you know (note that null candidates), which can be ("dog/Noun" -> term), in the given context of Text.

Example #2. `getRulesForLeft("dog", noun, aJCas, 16, candidates)` where candidates are: `[ ("pet", noun), ("animal", noun), ... (lemma_n, pos_n) ]`

- get me rules where the possible candidates are given as `candidates`, on the term dog, as appears with the context given in aJCas Text part pos 16.

Example #3. `getRulesForLeft(null, null, aJCas, 16, candidates)` where candidates are: `[ ("pet", noun), ("animal", noun), ... (lemma_n, pos_n) ]`

- get me rules where the possible candidates are given as `candidates`, that can be used in the aJCas pos 16. Note that this call does not specify lemma of left!

4.11.2.1.3. `getRulesRight` family added by `LexicalResourceWithContext`

- `List<LexicalRule<? extends I>> getRulesForRight(String lemma,PartOfSpeech pos, JCas context, int lemmaRBegin,List<Tuple<String,PartOfSpeech>> candidates) throws LexicalResourceException;`

Let's see some more examples for this case. Note that the methods does **not** care about Text side, and it only sees the Hypothesis side. Thus, the meaning of a call is "what are the lexical rules that can applied to the term in its own context?".

```
___0123456789012345678901234567890l  -- CAS SOFA position.
T: [ don't care ]
H: John took hit pet for a walk.
```

Example #1. `getRulesForRight("pet", noun, aJCas, 14, null)`

- get me all rules that you know (note that null candidates), which can be valid as (term -> "pet/Noun"), in the given context of the Hypothesis.

Example #2. `getRulesForRight("pet", noun, aJCas, 14, candidates)` where candidates is: `[ ("dog", noun), ("cat", noun), ... (lemma_n, pos_n) ]`

- get me rules where the possible candidates are given as `candidates`, on the target term "pet" as appears with the context given in aJCas Hypothesis pos 14.

Example #3. `getRulesForRight(null, null, aJCas, 14, candidates)` where candidates is: `[ ("dog", noun), ("cat", noun), ... (lemma_n, pos_n) ]`

- get me rules where the possible candidates are given as `candidates`, that can be used in the aJCas Hypothesis pos 16. Note that this call does not specify lemma of left!

4.12. Phrase level knowledge resource

TO BE ADDED

4.13. Annotator component

4.13.1. Description

As described in the section 3 (LAP and type system), EOP relies on CAS as the main data structure to keep various levels of annotations. From LAP level, they are generic language processing results like POS, lemma, syntactic dependencies, NER, SLR and etc. As described in the section, CAS type system can express (with new extensions and additions of types), and add almost any data related to the input pair.

The expressive power of CAS and its annotations are not limited to generic processing like parsing, or SLR. For example, consider a new annotation type that annotates partial entailment relation, or partial contradiction relations. Imagine that the component is able to determine partial contradiction with some confidence. (say, the time duration of the adverb phrase of the text cannot match another adverb phrase in hypothesis, etc). This type of information should be determined and passed to the total (full) entailment decision algorithm. If we want to express such data, in a common way, which would be the best within the platform? As annotations within CAS, is a natural data type for such a component. The component analyzes the given CAS, determines what the component is supposed to do, and add what is determined by the component in terms of specific annotations (e.g. in the above example, phrase level contradiction).

We define this kind of component as a separate type of component, and call it the Annotator component. An annotator component gets one CAS, analyzes the given CAS and add additional information in forms annotation. The added annotation gives extra information for EDAs, which would help EDA's decision on entailment.

What is the difference between a `AnnotatorComponent` and a LAP component? It seems that both adds annotations in CAS. The following list outlines the major differences. In general, we expect it would be easy to figure out where a new annotation adding module to be added.

- LAP components annotator provides generic annotations, such as POS, NER and syntactic structures. Annotator component provides more textual entailment specific annotations.

- LAP annotates each view separately, each as a text. Annotator component annotate the given CAS as a T-H pair (or pairs). Annotator component treats the CAS as entailment pair data, not just two set of textual input.

- LAP components belong to LAP. They cannot access CORE components like lexical or syntactic knowledge resources. Annotator component is part of CORE, and actively utilizes other CORE components.

We also define sub-types of Annotator component, to identify annotator components by their types and roles. In the version 2.0 of the specification, only the `AlignmentComponent` is the subtype of `AnnotatorComponent`.

4.13.2. Interface `AnnotatorComponent`

The interface defines only one method.

- `public void annotate(JCas aJCas)`: the method gets one JCas. Once the annotate process is done, the given JCas enriched by the added layer of annotations. If the annotator component was not able to add annotations (for example, needed LAP annotation like lemma was missing, or some failure of activating resources, etc), the component must raise a proper exception. If the call is successfully returned, it always means the process was successful --- even though there were no added annotations (e.g. contradiction annotator, but there was no contradiction).

4.14. Alignment component

4.14.1. Interface `AlignmentComponent`

Alignment component is a sub-type of annotator component. The component interface does not add any new method. This (empty) interface is defined as an independent interface mainly to serve the purpose of identifying alignment components among alignment components.

All components that implement `AlignmentComponent` **must** annotate according to the specific CAS types that are designed for alignment annotations within EOP, which are described in the following section.

4.14.2. CAS types for Alignment annotation

Spec 1.1 defined `AlignedText` CAS type. The type had been designed to express surface level of alignments (token, and phrase levels). The simple type was good for those, but it is not appropriate for aligning more complex annotations, such as nodes, trees, graphs, or any future data that will be added for EOP CASes. In this section, we define a new, more general annotation types. The new types are designed for alignment representation on any annotations.

The design goal for the new types are permitting maximum flexibility, while maintaining simple types that is generic enough. We define two types for this.

- `excitement.alignment.Target`
    - CAS annotation type that can point one or more annotations (tokens, parse nodes, NER nodes, or any annotations)
    - A list that groups annotations in one View.
    - The purpose is to allow dynamic alignment of material, including structures made by multiple annotations.
- `excitement.alignment.Link`

  - CAS type that links two Target.
  - Multi-view type: a Link connects one target in T (TextView), the other target in H (HypothesisView).

We make no assumptions regarding what annotations are aligned by Link and Target annotations. One Target can be linked by an arbitrary number of Link, also a Target can group an arbitrary number of Annotations. Note that uima.tcas.Annotation is the super type of almost all CAS annotation data. Since a Target can group Annotation, it can group any type of annotations in CAS.

You can find the formal definitions of the two CAS types in the following sections. Their definitions are pretty straightforward, but please take a look on "notes on usage" of each type for its begin-end behavior, before using the types.

4.14.2.1. alignment.Target

- Name: eu.excitementproject.type.alignment.Target
- is-a: uima.tcas.Annotation
  - inherits feature begin and end
- Features
  - targetAnnotations (FSArray [Annotations]): This is a FSArray that can hold one or more annotations. A target should mark one or more annotations
- Note on usage
  - begin holds the minimum begin value among annotations in targetAnnotations.
  - Likewise, end should point the maximum end among annotations.

4.14.2.2. alignment.Link

- Name: eu.excitementproject.type.alignment.Link
- is-a: uima.tcas.Annotation
  - inherits feature begin and end
- Features
  - textSideTarget (aligment.Target): this feature points one Target in TextView. Mandatory, and should not be null.
  - hypotehsisSideTaget (alignment.Target): this feature points one Target in HypothesisView. Mandatory, and should not be null.
  - type (String): a string that describe the type of the link. (e.g. "Lexical Alignment", "Dependency Sub-tree", etc). Note that we do not attempt to group types of different alignments. The type is merely a string, that identifies what kind of module added the link.
  - linkStrenth (Double): a double value that ranges between [-1, 1]. A plus-valued link means that this is a positive alignment. In TE context, this positive value means positive local entailment. The higher the value is, the more confident the annotator component is. Minus value means that this is a negative alignment. That is, the annotator is confident that this cannot be a local entailment (essentially means local contradiction).
- Note on usage

- Indexing of the `Link` within CAS: All annotations must be indexed to the CAS or CAS Views to be accessible. By convention we define that a `Link` instance should be indexed on `HypothesisView`. So iteration over the `HypothesisView` will get all alignment `Link` instances within the CAS.
- `begin` and `end`: Also as a convention, both span values should hold the same value to that of `hypothesisSideTarget`. Again, this is also for generalizing accessing of `Link` instances.

4.14.2.3. Note on the deprecated old type

Please note that old `AlignedText` type is deprecated, and no longer used. Even for surface level of alignments, `AlignmentComponent` must only use `alignment.Link` and `alignment.Target`.

4.15. Interface `Trainable`

4.15.1. Description of the interface

A RTE system generally requires the process of training: actual content of training can vary, but a working RTE engine requires various parameters, or knowledge to make a good entailment decision. Such parameters are normally acquired via a training data, and then they are stored in a model.

In the earlier versions of the specification (pre-2.0), the training capability of RTE engine was only defined on the top level, on EDA. EDA has `startTraining`, and `initialize`. The two methods works with configurable values (in the form of common configurations), and starts training with the given configuration and generate a new model(`startTraining`), or load a pre-existing model (`initialize`). This choice of top-level-only training is reasonable for systems where there is only one entry point for training. For example, TIE, EDITs and BIUTEE worked well with this setup. An instance of the EDAs only uses one model per EDA (per EDA configuration).

However, not all training behavior of RTE systems can be mapped with this way. An example is alignment-based approach. Alignment is an essential step for alignment-based RTE systems. This monolingual alignment process can be expressed as a component (as explained in annotation component) that can be used by multiple EDAs, or as of aligner module itself. Which means that trained parameters of the component is independent to a specific EDA, and it should be saved as its own model.

Our approach on "trainable component" is again minimal, similar to our definition of the training method on EDA. We define a common starting point for training. Each component that can produce a model and training capability should implement the `Trainable` interface. This interface provides a common entry point for initiating the training process. Each training process (and their required data) can vary greatly. Those varying options will be provided in the configuration.

Note that we do not define common model storage methods (e.g. what to save, where to save, what kind of format the model file has, etc). This is provided by each component, and each component that provides this interface must document the required configurable values in its JavaDoc documents. Thus, this interface only provides common starting point for training. Although this is a minimal contract, the interface enables several common behavior.

- All trainable components are recognizable with this interface.
- Each trainable component shares the same starting point. Thus, when an EDA wants to train multiple `Trainable` component instances, it can do so with one valid common configuration and iterating over trainable components by calling the method provided by `Trainable` interface.
- The interface forces all `Trainable` to support Common-config.

4.15.2. Methods of the interface

The interface only provides one method.

```
void startTraining(CommonConfig configuration)
```

Calling this method makes the component to start the training process, as defined in the configuration values. Once the training process is finish, the method will return without any return value, but the training result will be written in a designated position in the configuration.

Note that once `startTraining()` is finished, the component instance does not need to comeback to "prediction (or work)" mode. For example, let's assume we have an alignment component. Normal (prediction) mode call would be `annotate()` method as defined in the alignment component. However, Once `startTraining()` method is called and model is saved via the method, the instance do not need to be able to process further `annotate()` call. Component writers can consider `startTraining()` as a one-way mode change command for the component to training mode. This assumption is added to make actual implementation of trainable components simpler.

Also note that we do not provide `initialize()` method in this interface, in contrast to `EDABasic` interface.

4.16. Semantic Similarity Decision Algorithm

4.16.1. Description of the component

4.16.1.1. Semantic Textual Similarity task

Semantic Textual Similarity (STS) is a task of determining the semantic similarity between a pair of textual data. The goal is described as "given two textual input, find out the degree of semantic equivalence between the two inputs." (http://ixa2.si.ehu.es/sts/) The task is related to both Textual Entailment and Paraphrase, but it is different in a way. Unlike Textual Entailment, STS task is not directional. For example, in TE, "a car" entails "a vehicle", but not the other way around. In STS, it is a symmetric relation. Also, the result of STS is graded similarity, not categorical decision (e.g. the pair [riding a vehicle, riding a car] is more similar than [riding a wave, riding a car]).

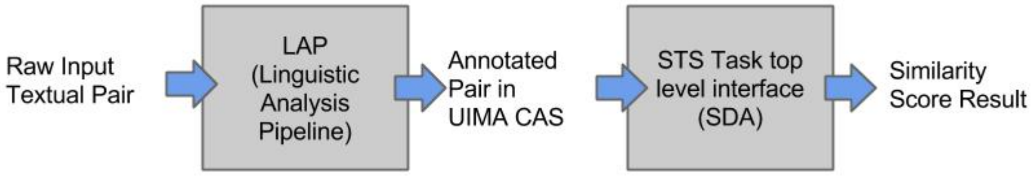
STS task has been evolving with SemEval tasks, since 2012. (xref SemEval-2012 Task 6). The task sets provide concrete training/test set similar to RTE dataset. In the task data, the similarity degree is represented in a (continuous) scale of real number between 0.0 to 5. Degree of 0 means there is no relation at all, while degree of 5.0 represents semantic equivalence.

STS is a separate task where its goal is different to TE task. However, there are many potential of possible common contributions. For example, the assumption of EDITS EDA is actually based on similarity (e.g. if T and H are not related, they can't be entailment), and many RTE systems and STS systems can actually share various common processing components (e.g. distance components) and/or static components (e.g. knowledge resources). We believe that supporting STS systems within EXCITEMENT open platform has potential of future collaboration and sharing of useful components and modules that might benefit both communities.

4.16.1.2. Supporting STS system in EXCITEMENT platform

In the specification version 2.0, we start supporting STS systems by first adopting the top level interfaces. By top level, it means the same level as EDA level. The following figure shows the conceptual drawing of where the newly defined top-level STS module lies.

Figure 12. Top level data flow of SDA


Raw Input Textual Pair
LAP (Linguistic Analysis Pipeline)
Annotated Pair in UIMA CAS
STS Task top level interface (SDA)
Similarity Score Result

The figure is very similar to the top level figures for `EDABasic` interface. Text-pair is given to LAP, and the LAP generates CAS output which holds the annotated pair. However, instead of EDA, here comes the newly defined SDA (Similarity Decision Algorithm). The SDA represents the STS task in the top-level abstract, where the input is annotated pair data (in CAS), and the output is new Similarity Score result data type that represents STS similarity output (`SemanticSimilarityDecision`).

One interesting aspect here is that we reuse the same CAS and same data types. Note that some names in CAS assumes TE-style inputs (for example, view names are designated as "Text", and "Hypothesis"), but LAP-output is generally (almost always) neutral to the task. STS systems simply can consider "Text" of input pair as "s1 (sentence 1)", and "Hypothesis" part as "s2 (sentence 2)". Note that most of the LAP annotations are not aware of two views (text and hypothesis), and task-neutral.

Thus, CAS for EDA and SDA are almost identical. The only difference on CAS level is the gold annotation that is required for training part. EDA uses `gold` value annotated with `EntailmentPair` annotation type. However, SDA does not utilize this gold value. Instead, there is a new CAS type defined for SDA. `SemanticSimilarityPair`, which is an extension of `EntailmentPair`, and holds `goldSimilarityValue` that scales between `0` to `5`. Note that We choose to extend `SemanticSimilarityPair` from `EntailmentPair`, so on the processing side (once the model is learned), SDA can share CASes that are prepared for EDAs without any modifications.

4.16.2. Interface SDABasic

```
public interface SDABasic<T extends SemanticSimilarityDecision>
```

This interface supports basic capabilities for STS task. It defines 4 methods, similar to `EDABasic` interface.

- `public void initialize (CommonConfig config)`: This method will be called by the top level programs as the signal for initializing the SDA. All initialization of an SDA like setting up sub-components and connecting resources must be done in this method. An SDA 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.

- `public T process (JCas aJCas)`: This is the main access point for the top level. The top level application can only use this method when the SDA is properly configured and initialized. Each time this method is called, the SDA should check the input for its compatibility. Within the EXCITEMENT platform, SDA implementations are decoupled with linguistic analysis pipelines, and you cannot blindly assume that CAS input is valid. SDA implementation must check the existence of proper annotation layers corresponding to the configuration of the SDA. The similarity decision is returned as an object that extends `SemanticSimilarityDecision` interface which essentially holds the similarity as a float value.

- `public void shutdown()`: This method provides a graceful exit for the SDA. This method will be called by top-level as a signal to disengage all resources. All resources allocated by the SDA should be released when this method is called.

- `public void startTraining(CommonConfig c)`: startTraining interface is the common interface for EDA training. The interface signals the start of the training with the given configuration c.

4.16.2.1. Type `SemanticSimilarityDecision`

This interface represents the return value for SDA `process()` results. It only holds the basic information, and SDA implementers are expected to extend it to hold additional information for their SDA. It defines two methods.

- `public double getSimilarityValue()`: This method returns the similarity value decided by the SDA.

- `public String getPairId()`: This method returns Pair id as described in the CAS.

4.16.2.2. Limitation of STS task support in the specification version 2.0

Please note that for the moment (ver. 2.0), we only have top-level STS support only. The specification 2.0 is yet to resolve the component reusing issue. It is not so simple to actually reuse EXCITEMENT core components in SDA. This is because of the fact that many component types implicitly assumes T -> H directional processing. For example, Lexical resources and Syntactic resources are asymmetric from its very definition. (LHS entails RHS.) It is also similar for Distance components: although this is not as explicit as knowledge resources, but they also assumes directed distance from T to H (e.g. edit distance from T to H with asymmetric calculation).

How do we want to define and harmonize components to be reused in both EDA and SDA? For example, do we want to additionally define symmetric versions of lexical resources? This is an open issue without easy answers. In the specification version 2.0, we do not try to resolve this issue yet. For the moment, we only provide top level for STS with SDA interface.

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 5.1.6, “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. 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.

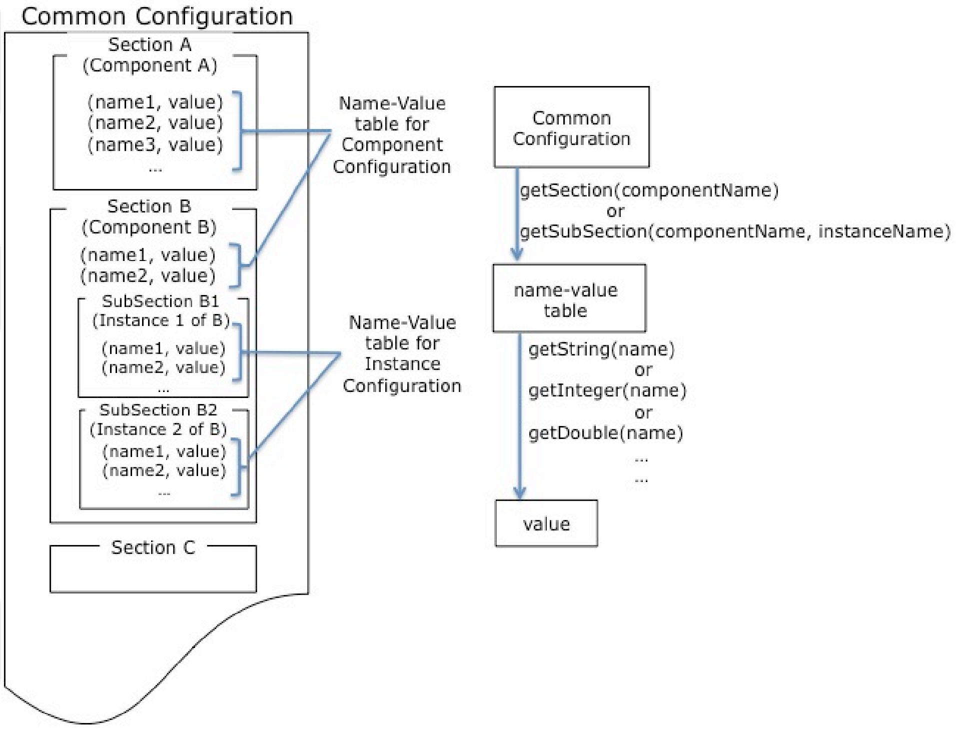
- Storage in XML format: The human-readable version of the common configuration will be stored and read as XML format. Users can edit the XML file as ordinary XML file. Since names are supposed to be unique, a name collision leads to an exception when the XML serialization of the configuration object 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 constructor and/or `reconfigure()` methods. Each component must read the configuration and initialize and/or reconfigure itself accordingly.

Figure 13, “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 13. 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 an XML file as a string) can be accessed with `getString(name)`, `getInteger(name)`, or `getDouble(name)`. The XML strings will be converted automatically into the requested type.

As an in-memory object, all name-value pairs are simply seen as a pair of 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 code is given only as an example. The specification does not assume anything about how the name-value pairs are represented in XML.)

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).

Instance configuration areas (SubSections) are only needed for the components that support multiple instances with different configurations. If all instances of a component have the same configuration, the component do not use instance configuration and uses only the component configuration. For example, knowledge components generally have no need to have different configuration even if they have multiple instances.

However, some components must have multiple instances with different configurations. Let's assume there is a distance calculation component that is shipped with two parameters. One is a `path` to a static resource (say, a dictionary), and the other is a numeric parameter `alpha`. Assume that the EDA needs to run two instances: one with alpha as 1.0, the other with alpha as 2.0. By having instance-wise configurations, the EDA can run two instances of the distance calculation component, each with different configuration. In this example, parameter `path` will stay in the component section, and parameter `alpha` will be written in the instance subsection.

Note that there is 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. (In the above example, parameter `path` is not visible to the name-value table of the instance subsection.)

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 the current version.)

5.1.3.1.1. constructor `CommonConfiguration()`

`CommonConfiguration(File)` Common configuration provides a constructor that loads a configuration from an XML File. If the target file is not valid (invalid XML or missing mandatory section), the call will raise an exception. CommonConfiguration will also provide a constructor with empty arguments, which can be used to generate a new configuration file.

5.1.3.1.2. 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.

5.1.3.1.3. method `getSubsection()`

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

5.1.3.1.4. method `saveConfiguration()`

`void saveConfiguration(File)` This method saves current configuration to an XML file. It will save whole values as a single XML file. Note that this save method should provide a safety mechanism that will prevent overwriting existing XML configurations (ie. only permitting generating new configuration files, etc).

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.

5.1.3.1.5. method `getConfigurationFileName()`

`String getConfigurationFileName()` This method is a convenience method that returns the full path name of the current configuration file. Will return null, if the configuration is not originated from a file and never saved before.

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)

public File getFile(String name)

public File getDirectory(String name)

...
```

Configuration data is stored as text strings in the XML files. XML parsers can recognize XML primitive types like string, boolean, decimal, double, etc. In this specification, we only show get methods for a few basic types, like string, integer, double and File. Note that the actual implementation can provide many more (like get methods that will return enum, list of enums or other basic types, etc), depending on the need of the Component writers.

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 the requested type. If the conversion fails, the get method will raise a conversion exception (one of ConfigurationException).

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)

public setFile(String name, File value)

public setDirectory(String name, File 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. Common Configuration XML file format

This section first outlines the adopted XML configuration format with an example. Then, it iterates a few issues around configuration names and GLOBAL section (PlatformConfiguration).

Let's first see an example file that follows the proposed file format. It has a few sections (components) within it. Each name-values are represented with a `property` element. The element has one attribute (`name`), and actual value is written as element value within the property element.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE configuration [
<!ENTITY myVar "Some common #PCDATA that can be reused... ">
]>

<configuration>
<section name="PlatformConfiguration">
<property name="activatedEDA">core.MyEDA</property>
<property name="language">EN</property>
</section>

<section name="PhoneticDistanceComponent">
<property name="KlingonDictionaryPath">$RESOURCE/VD/KDict.cvs</property>
<property name="beta">0.1</property>

<subsection name="instance1">
<property name="consonantScore">1.0</property>
<property name="vowelScore">0.6</property>
<property name="alpha">0.17663311</property>
</subsection>

<subsection name="instance2">
<property name="consonantScore">0.6</property>
<property name="vowelScore">1.0</property>
<property name="alpha">0.17663311</property>
</subsection>

</section>

<section name="core.MyEDA">
<property name="myLongKey">&myVar;</property>
<property name="someOption">PhoneticDistanceComponent,
                       EditDistanceComponent</property>
</section>
```

```
</configuration>
```

Note that,

- All section names must be unique (globally).
- All subsection names must be unique within the section.
- All property names are unique in each table (section/subsection)

If any violation is observed (including those and possible others), CommonConfig must raise an exception while loading the file.

A small note on configuration value names and `PlatformConfiguration` (Global) region:

- `PlatformConfiguration` section: This global section of the common configuration holds some configuration data that will be shared among all components and EDAs. For the first prototypes, we will only adopt two values. `activatedEDA` holds the name of the EDA that is selected in this configuration. `language` holds the language of the current configuration.
- Default configuration: Each EDA implementer must provide a suitable default configuration, so the user can start using the EDA with relative ease.
- Names within a section (component): Each name within each component configuration section is independent. The component implementer can use the name that is suitable for the module. In the future, we might iterate over configuration names, and we will try to normalize common names, if such need arises.

5.1.5. Extending the common configuration features

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

5.1.6. 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*.

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
          channel CDATA #IMPLIED>
<!ELEMENT pair (t,h)>
<!ATTLIST pair
          id CDATA #REQUIRED
          entailment (ENTAILMENT|NONENTAILMENT|CONTRADICTION|
                          PARAPHRASE|UNKNOWN) #REQUIRED
          task (IR|IE|QA|SUM) #IMPLIED >
<!ELEMENT t (#PCDATA)>
<!ELEMENT h (#PCDATA)>
```

Now the top element `entailment-corpus` has an optional attribute `lang` and `channel`. 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. This is the same for `channel` value. The value records the input channel (source of the input like e-mail, transcribed phone call, etc), and it is an optional value. Also, the possible values of entailment relationship are extended to match those of Section 4.2.1.6, “enum `DecisionLabel`”.

Support for this xtended RTE-5 format is mandatory. EDA implementers must support processing of the 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. Linguistic Analysis Pipeline

6.1. The goals of the LAP interface

LAP (Linguistic Analysis Pipelines) is an important part of the platform that provides all linguistic analysis capabilities for the EXCITEMENT platform and its users. It provides not only normal NLP annotation capabilities like tagging and parsing, but also the capability to generate well-formed CASes for EXCITEMENT entailment decision algorithms (EDAs).

The purpose of this set of LAP interfaces is twofold. First, it defines the common minimal capability that each pipeline should provide, in terms of interface definition. Each EDA implementer must provide a compatible pipeline for the EDA. For the developer who will provide a new pipeline, these interfaces can serve as an implementation guideline.

The second purpose of the interface is to make the underlying pipeline mechanism transparent for the end user. As stated in the previous sections (Section 3.2.3, “Adoption of UIMA in EXCITEMENT”), the EXCITEMENT consortium already agreed on using UIMA CAS. However, we have not yet come to a decision about the adoption of UIMA runtime mechanism (like AE, AAE or CPE, etc). In the first iteration of the platform, it is expected that some of the pipelines are composed by a set of UIMA AEs, while other pipelines will only "translate" their pipeline output to the UIMA CAS.

In this situation, it is important to define a coherent set of interfaces that is shared by all pipelines, regardless to the underlying mechanism. The users should be able to call all different pipelines with the same argument and signatures. This is more important for users who just want to use the platform only as "off-the-shelf TE engine". For example, knowledge of CAS or UIMA types should not exposed to such users. Those users should be able to use LAP pipelines for their EDA calls without knowing what CAS has within it.

The following sections define the three set of methods that will be shared by all pipelines.

6.2. Interface `LAPAccess`

`interface LAPAccess extends Components`

This interface defines the minimal set of methods that a working LAP pipeline should provide for EXCITEMENT platform. It defines three types of methods. One for generic annotation, one for single pair online-generation, and one for collection processing of single-pair input files.

6.2.1. method `generateSingleTHPairCAS`

- `public JCas generateSingleTHPairCAS(String text, String hypothesis) throws LAPException`
  - Arguments: `text` holds the text string of the single pair TE problem. `hypothesis` holds the hypothesis string of the problem.
  - returns: a `JCas`, where the CAS is ready to be used as an input for an EDA.

This method gets two strings as input and generates a CAS for an EDA. The resulting CAS must be a well-formed CAS that contains all types can are necessary for a complete description of the entailment problem (i.e. all types from the `EXCITEMENT.entailment` namespace), as defined in Section 3.3.4, “Additional Types for Textual Entailment”.

The amount of linguistic annotation can differ among implementations (say, EnglishTokenizationOnly(t,h) only returns token with TE annotations, while GermanParsingAndNER(t,h) returns tagging, parsing, POS and NER, etc).

6.2.2. method `processRawInputFormat`

- `public void processRawInputFormat(File inputFile, File outputDir) throws LAPException`
  - Arguments: `File inputFile` holds the raw input XML file. `File outputDir` holds the output directory where the resulting CAS XMI should be stored.

This method gets two `File` as arguments. First argument is a single input file that follows the raw input XML format (as defined in Section 5.2, “Input file format ”), and the second argument `outputDir` is the output directory. The method returns nothing. Analysis results are stored as serialized CAS files in the directory. (standard XMI serialization of UIMA CAS).

Again, the generated CASes should be well-formed and must contain all types can are necessary for a complete description of the entailment problem (i.e. all types from the `EXCITEMENT.entailment` namespace).

6.2.3. method `addAnnotationsToCAS`

- `public void addAnnotationOn(JCas aJCas) throws LAPException`
  - Arguments: `aJCas` holds the CAS object that will be annotated by the analysis pipeline.
  - Returns: Nothing. The JCas argument is enriched by the linguistic analysis annotations provided by the pipeline.
- `public void addAnnotationOn(JCas aJCas, String viewName) throws LAPException`
  - Arguments: `aJCas` holds the CAS object that will be annotated by the analysis pipeline. This overridden method is for multi-view CASes. `aJCas` holds two or more view, where one of them has the view name equal to `viewName`.
  - Returns: Nothing. The argument JCas is enriched: one of its view where its name is equal to `viewName` is enriched by the linguistic analysis annotations provided by the pipeline.

The interface is a generic interface that can be called for adding linguistic annotations for any data. `addAnnotattionOn` method does not produce CASes that are ready for EDAs. It only adds generic language analysis results to the provided CAS. This method provides users with the ability to analyze some text data that is not directly a TE pair.

The method's argument is not a string or text, but a JCas. The user must have some knowledge on CAS to use this interface. For example, they needs to know how to generate a new CAS and add text data to it, before calling this method. This design is intentional. The user needs to know about CAS data access methods beforehand, to open up and manually access the analysis results provided by a pipeline. Moreover, all entailment core interfaces that accept CAS as an argument accept only the EDA-compatible CASes. That is, a specific form and type of CASes that can be consumed by EDAs and components (CASes with `EXCITEMENT.entailment.*` types, and with `TextView` and `HypothesisViews`).

Thus, the `addAnnotationOn` interface (unlike `generateSingleTHPairCAS` or `processRawInputFormat`) produces "generic CASes", which can be non-EDA-compatible CASes. The resulting CASes of `addAnnotationOn` are meant to be used mainly by the caller himself, when they require to perform linguistic analysis on additional data of some kind. For example, the Transduction layer needs to use this functionality to determine which sentence of a customer e-mail can be regarded as text or hypothesis, respectively.

Note that `addAnnotationOn` exists in two versions: one that applies to the single view CAS and one that applies just a specific View within the CAS. The pipeline implementer must provide both of them; we recommend to implement `addAnnotationOn(JCas)` simply as a call of addAnnotationOn(JCas,"_InitialView") (single view CAS is a CAS with one view, that is implicitly named as "$_{\text{InivitalView}}$").

addAnnotationOn should not add any type that belongs to EXCITEMENT.entailment.* types (types related to text, hypothesis, pair and entailment metadata).

An LAPAccess implementation must provide all three sets of methods defined in the interface.

6.3. Room for improvements with UIMA adoption

There is a room for improvement for the interface, if the consortium decides to adopt UIMA Analysis Engines (AEs) as the basis for LAP components. Such improvements (such as using UIMA AE, AAE, and runtime to share and standardize LAP components) will be considered fully, after the project members have decided UIMA adoption as scheduled in Section 3.2.3, “Adoption of UIMA in EXCITEMENT”.

7. Further Recommended Platform Policies

7.1. Coding Standard

This section describes the coding standard for the EXCITEMENT platform. You can also find the latest coding standard from the project Wiki: EOP Wiki: Coding Guidelines [https://github.com/hltfbk/Excitement-Open-Platform/wiki/Coding-Guidelines]

7.1.1. 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.

7.1.2. 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.

7.1.3. 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.

7.1.4. 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.

7.1.5. Code annotations

EXCITEMENT platform will adopt some Java code annotations, starting with the followings.

- `@LanguageDependent`: Every class that is language dependent (e.g. only for English, only for Spanish, etc) should be annotated by `@LanguageDependent`.

- `@ThreadSafe`, `@NotThreadSafe`: These two annotations are for explicitly marking thread safety of a class.

Actual definitions of the Java annotations and their usage will be provided by the implementation team (WP4). Also, the list of the annotations will be expanded along with the implementation effort.

7.2. List of Exceptions

Exceptions are part of the method signature, similar to inputs and return types. However, it is not easy to specify an exception hierarchy for the EXCITEMENT platform in detail at this stage. Thus, in this specification, we will only try to outline them. The objects outlined in this section will form the top level objects in the EXCITEMENT exception hierarchy. More detailed use cases and additional exceptions will be added in the future during the development in WP4.

The following exceptions form the top layer of the exception hierarchy. They directly inherit from java.lang.Exception (or a putative future generic EXCITEMENT 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 and implementations of EDAs can generate this type of exception. All checked exceptions thrown from EDA code should use or inherit this exception.

- `ComponentException`: This is 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`: Implementations of knowledge components (lexical resource and syntactic resource) can raise this exception.

- `DistanceComponentException`: Implementations of distance calculation components can throw this exception.

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

7.3. Common Logging for Entailment Core

Components often need to expose their progress and internal logic for various purposes. EXCITEMENT core components are no exceptions. For this purpose, all core components (including EDA) will share a common logging facility. The `CommonLogging` class must support the following capabilities:

- Priority: the logging capability must support different levels of priorities. For example, TRACE, DEBUG, INFO, ERROR, etc.

- Originator: The originator of a log message should be identifiable. The common logging must provide a principled way of distinguishing the log originator, with component name (`getComponentName` of `Components`) and/or class name.

The specification does not specify further detail of the common logging capability. It is expected that one of the well known Java logging libraries will be imported for the EXCITEMENT platform.

Note that, at least for the first iteration, the entailment core common logging will only cover the EDAs and core components, not LAPs. For LAP, we will use UIMA logging facilities. Integration of two logging system will be considered in the second iteration. Also note that our coding standard does **not** permit usage of standard error output Section 7.1.2, “Error Handling ”. Thus, the log output is the only recommended way of exposing the inner progress of a component.

7.4. 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 summarizes 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 the naming of constants (like `PI_VALUE`), where uppercase letters 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 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 object. UIMA feature structures are equivalent to class members, and all features names should start with a lowercase letter, 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 letter.

7.5. Boundary of LAP and Entailment Core: Where should my new component go?

Another issue that is related to UIMA and the platform concerns the boundary between the LAP (which uses CAS to represent its outputs) and the Entailment Core (which uses proprietary Java data structures). Which components should be provided as UIMA component (LAP component that outputs to CAS), and which components should be provided as entailment core components (with Java APIs)?

At the beginning of the project, only common linguistic analysis components were expected to be provided as LAP component. However the development is towards packaging TE-specific components like a "predicate-truth annotator" as a LAP component as well. This gives rise to the question where a new component should be conceptualized when we want to add a new capability: as a new UIMA (LAP) component? Or, alternatively, as a new entailment-core component?

The following recommendations represent a "best practice" produced collectively by the academic partners.

- Generic language analysis components must be provided as LAP components, since they will be also by other layers (transduction layer, and user level code, etc).
- LAP components (i.e. UIMA annotators) are best suited for annotation (adding labels to portions of text). If an analysis components main goal is to add annotations, it should be implemented as a LAP component.
- Given we now have text and hypothesis markings also as UIMA CAS annotations, it is possible to add annotations that are specific for textual entailment.
- It is recommended that a new analysis component should be delivered as a LAP component if it is *natural* to represent is as an annotator (for some interpretation of "natural"). The advantage of doing so is that it can add new analysis capability *without* the need to define additional APIs.
- A final consideration is serialization. UIMA provides a generic procedure for storing CAS objects. Thus, components that create *static* annotations -- that is, which can be considered to have an independent status as an annotation layer of a dataset, independent of one particular experiment -- can be conceptualized well as LAP components and can ignore all considerations about serialization.

Note that the above list is not a hard guideline, but closer to a suggestion. It is not very hard to imagine good counter examples for each point. For example, distance calculation components are not natural to be represented as an annotator. But a developer might want to represent it as a LAP module when he wants to store the result statically in a CAS to speed up repeated experiments.

Thus, developers have a certain level of freedom in their choice between implementing components with the LAP or the EC. As we grow more confident on UIMA and common platform, we may be able to provide more concrete best practices on this issue.

8. References

Bibliography

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

[BIUTEE] EXCITEMENT WP3 ITD subgroup, Major Components of BIUTEE - Bar Ilan University Textual Entailment Engine, 2012. *https://dl.dropbox.com/u/13718630/reports/BIUTEE_component_overview.pdf*

[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*

[EDITS] EXCITEMENT WP3 ITD subgroup, Major Components of EDITS - Textual Entailment Engine of FBK, 2012. *https://dl.dropbox.com/u/13718630/reports/EDITS_component_overview.pdf*

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

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

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

[UIMAWS2013] Noh and Pado, Using UIMA to Structure an Open Platform for Textual Entailment, UIMA workshop at GSCL, 2013. *Using UIMA to Structure an Open Platform for Textual Entailment [http://ceur-ws.org/Vol-1038/paper_3.pdf]*

[JNLE2013] Pado et al, Design and Realization of a Modular Architecture for Textual Entailment, Natural Language Engineering, 2013. *Design and Realization of a Modular Architecture for Textual Entailment [http://www.nlpado.de/~sebastian/pub/papers/jnle13_pado.pdf]*

[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*

[TIMEX3] TimeML Working Group, Guidelines for Temporal Expression Annotation for English for TempEval 2010, August 14, 2009 *Timex3guidelines-072009.pdf [http://www.timeml.org/tempeval2/tempeval2-trial/guidelines/timex3guidelines-072009.pdf]*

[TIE] EXCITEMENT WP3 ITD subgroup, Major Components of TIE - Textual Entailment Engine of DFKI, 2012. *https://dl.dropbox.com/u/13718630/reports/TIE_component_overview.pdf*

[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]*

Appendix A. Type Definition: types for general linguistic analysis

Starting from the specification version 1.1, we have attached a java project to the specification. You can download the attachement that holds all the interface codes and the type system *EXCITEMENT SPEC 1.1 interfaces and type systems* [https://dl.dropbox.com/u/13718630/codes/excitement_interface_codes_aug29.zip]

A.1. Segmentation types

You can find the type definition in XML at `LexicalUnits.xml` among the attached project files.

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.Annotation`
    - 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.Annotation`
    - 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.Annotation`
    - 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

You can find the type definition in XML at `POS.xml` among the attached project files.

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.Annotation`
  - 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 also some extended POS tags designed to work for specific text domains. 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 code that does not understand the extensions, is still be able to access them in terms of the generic types 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. NER types

You can find the type definition in XML at `NamedEntity.xml` among the attached project files.

These types are related to named entity recognition. It uses a top type, which represents a generic entity. Actual types (person, organization, 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: (From BBN) This type is named after its subtypes, nationality, other, religion, political. The distinction between NORP and other types is morphological. American and Americans is a nationality, while America and US are geographical entities.
  - 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.4. Types for Constituency Parsing

You can find the type definition in XML at `Constituent.xml` among the attached project files.

These types are related to constituency parse results. There is a top constituency type that represents the constituency parse tree node. Various nodes are represented as subtype of this top node type. Only the top constituency type `Constituent` has features, and all subtypes are inheriting the features.

- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent
    - Description: The type represents a parse tree node. Actual nodes are represented by the subtypes of this type. This type provides the features. It has four features. `constituentType` denotes the output of the parser (raw output of the node type, the type is mapped into one of the subtype of the node). `parent` denotes the parent of a parse tree node, and `children` keeps

    children of the nodes, as an array. The final feature is `syntacticFunction`, which is a string, that keeps syntactic function string for the node if the parse outputs a syntactic function value for the node.

  - Supertype: `uima.tcas.Annotation`
  - Features
    - `constituentType` (`uima.cas.String`)
    - `parent` (`uima.tcas.Annotation`)
    - `children` (`uima.cas.FSArray` of `uima.tcas.Annotation`)
    - `syntacticFunction` (`uima.cas.String`)

- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.DT
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.EX
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.FRAG
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.FW
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.IN
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.INTJ

- Description:
- Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
- Features: (no features)

- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.JJ
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.JJR
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.JJS
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.LS
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.LST
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.MD
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NAC

    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NN
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NNP
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NNPS
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NNS
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NP
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.NX
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.PDT

    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.POS
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.PP
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.PRN0
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.PRP
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.PRPP
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.PRT
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.PUNC

    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.QP
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.RB
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.RBR
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.ROOT
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.RP
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.RRC
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.S

  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.SBAR
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.SBARQ
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.SINV
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.SQ
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.SYM
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.TO
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.UCP

    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.UH
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.VB
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.VBD
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.VBG
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.VBN
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.VBP
    - Description:
    - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.VBZ

- Description:
- Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent
- Features: (no features)

- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.VP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.WDT
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.WHADJP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.WHADVP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.WHNP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.WHPP
    - Description:
    - Supertype: de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent
    - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.WP

  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)

- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.WPP
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.WRB
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.X
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.XS
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)
- de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.EX
  - Description:
  - Supertype: `de.tudarmstadt.ukp.dkpro.core.api.syntax.type.constituent.Constituent`
  - Features: (no features)

A.5. Types for Dependency Parsing

You can find the type definition in XML at `Dependency.xml` among the attached project files.

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

You can find the type definition in XML at `Coref.xml` among the attached project files.

Coreference resolution 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

You can find the type definition in XML at `SemanticRole.xml` among the attached project files.

- EXCITEMENT.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.
- EXCITEMENT.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.

A.8. Types for Temporal/NER events

You can find the type definition in XML at `TemporalExpression.xml` among the attached project files.

- `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 is the top annotation for temporal expressions, with a normalized time representation. It has four subtypes, which reflects TIMEX3 temporal types of Date, Time, Duration and Set [TIMEX3].
  - 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.
- `EXCITEMENT.temporal.Date`
  - Description: This type represents a temporal expression of type Date. The expression describes a calendar time like yesterday, November 1943, Tuesday 18th, etc.
  - Supertype: EXCITEMENT.temporal.TemporalExpression
  - Features: (no features)
- `EXCITEMENT.temporal.Time`
  - Description: This type represents a temporal expression of type Time. The expression refers to a time of the day, even if in a very indefinite way: eleven in the morning, late last night, twenty after twelve, etc.
  - Supertype: EXCITEMENT.temporal.TemporalExpression
  - Features: (no features)
- `EXCITEMENT.temporal.Duration`
  - Description: This type represents a temporal expression of type Duration. The expression describes a duration with explicit durations: 2 months, 48 hours, all last night, 20 days, etc.
  - Supertype: EXCITEMENT.temporal.TemporalExpression
  - Features: (no features)
- `EXCITEMENT.temporal.Set`
  - Description: This type represents a temporal expression of type Set. For example, every winter, each Monday, etc. It indicates a recurrence pattern.
  - Supertype: EXCITEMENT.temporal.TemporalExpression
  - Features: (no features)

A.9. Types for Text Alignment

You can find the type definition in XML at `TextAlignment.xml` among the attached project files.

- `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.
  - `alignmentType` (`uima.cas.String`): This feature holds additional information for the alignment as a string.

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 interpolated with an actual package name in the implementation.

Starting from the specification version 1.1, we have attached a java project to the specification. You can download the attachement that holds all the interface codes and the type system *EXCITEMENT SPEC 1.1 interfaces and type systems* [https://dl.dropbox.com/u/13718630/codes/excitement_interface_codes_aug29.zip]

B.1. Types related to entailment problems

You can find the type definition in XML at `EntailmentTypes.xml` among the attached project files.

- `EXCITEMENT.entailment.EntailmentMetadata`
  - Description: This type provides metadata for entailment problem.
  - Supertype: `uima.tcas.Annotation`
  - Features
    - `language` (`uima.cas.String`): This string holds the language of the entailment problem.
    - `task` (`uima.cas.String`): This string holds the task description which can be found in the RTE challenge data.
    - `channel` (`uima.cas.String`): This feature 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.
    - `TextDocumentID` (`uima.cas.String`): This field can hold a string that identifies the document of the TextView. This feature must have a value, if TextCollectionID is not null.
    - `TextCollectionID` (`uima.cas.String`): This field can hold a string that identifies the collection name where the document of the TextView belongs to.
    - `HypothesisDocumentID` (`uima.cas.String`): This field can hold a string that identifies the document of the HypothesisView. This feature must have a value, if HypothesisCollectionID is not null.
    - `HypothesisCollectionID` (`uima.cas.String`): This field can hold a string that identifies the collection name where the document of the HypothesisView belongs to.
- `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

You can find the type definition in XML at `PredicateTruth.xml` among the attached project files.

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

Appendix C. Entailment Core Interfaces

Starting from the specification version 1.1, we have attached a java source code project to the specification. You can download the attachement that holds all the interface codes and the type system *EXCITEMENT SPEC 1.1 interfaces and type systems* [https://dl.dropbox.com/u/13718630/codes/excitement_interface_codes_aug29.zip]

Instead of listing the source codes, Appendix C now points the source code file names within the attached Java project.

C.1. `interface EDABasic` and related objects

C.1.1. `interface EDABasic`

See `EDABasic.java` in `eu.excitement.entailment.core`.

C.1.2. `interface TEDecision`

See `TEDecision.java` in `eu.excitement.entailment.core`.

C.1.3. `enum DecisionLabel`

See `DecisionLabel.java` in `eu.excitement.entailment.core`.

C.2. `interface SinglePairProcessHelper`

See `SinglePairProcessHelper.java` in `eu.excitement.entailment.core.helpers`.

C.3. `interface EDAMulti*`

C.3.1. `interface EDAMultiT`

See `EDAMultiT.java` in `eu.excitement.entailment.core`.

C.3.2. `interface EDAMultiH`

See `EDAMultiH.java` in `eu.excitement.entailment.core`.

C.3.3. `interface EDAMultiTH`

See `EDAMultiTH.java` in `eu.excitement.entailment.core`.

C.4. `class MultipleTHModeHelper`

See `MultipleTHModeHelper.java` in `eu.excitement.entailment.core.helpers`.

C.5. `interface Components`

See `Components.java` in `eu.excitement.entailment.core`.

C.6. interface DistanceCalculation and related objects

C.6.1. interface DistanceCalculation

See DistanceCalculation.java in eu.excitement.entailment.core.component.distance.

C.6.2. class DistanceValue

See DistanceValue.java in eu.excitement.entailment.core.component.distance.

C.7. class LexicalResource and related objects

C.7.1. class LexicalRule

See LexicalRule.java in eu.excitement.entailment.core.component.lexicalknowledge.

C.7.2. class PartOfSpeech

See PartOfSpeech.java in eu.excitement.entailment.core.representation.parsetree.

C.7.3. interface RuleInfo

See RuleInfo.java in eu.excitement.entailment.core.component.lexicalknowledge.

C.7.4. interface LexicalResource

See LexicalResource.java in eu.excitement.entailment.core.component.lexicalknowledge.

C.8. interface SyntacticResource and related objects

C.8.1. class SyntacticRule

See SyntacticRule.java in eu.excitement.entailment.core.component.syntacticknowledge.

C.8.2. class BasicNode and related classes

See AbstractNode.java, BasicNode.java, NodeInfo.java, EdgeInfo.java, Info.java, and other codes in eu.excitement.entailment.core.representation.parsetree.

C.8.3. interface SyntacticResource

See SyntacticResource.java in eu.excitement.entailment.core.component.syntacticknowledge.

Appendix D. Supported Raw Input Formats

Note that only the following extended RTE-5 format is mandatory (EDAs should support them), and other formats are optional (EDAs may support them). See Section 5.2, “Input file format ”.

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

Appendix E. List of changes per version

TO BE ADDED (copy CHANGES file from GitHub repository