

**EXploring Customer Interaction via Textual EntailMENT**

**Deliverable 6.2: Textual inference   
components development, III cycle**

**Authors:** Kathrin Eichler, Lili Kotlerman, Vivi Nastase, Tae-Gil Noh

**Dissemination Level:** Confidential

**Date:** September 10, 2013

|  |  |  |
| --- | --- | --- |
| Grant agreement no. | | 287923 |
| Project acronym | | EXCITEMENT |
| Project full title | | Exploring Customer Interaction via Textual entailMENT |
| Funding scheme | | STREP |
| Coordinator | | Moshe Wasserblat (NICE) |
| Start date, duration | | 1 January 2012, 36 months |
| Distribution | | Confidential |
| Contractual date of delivery | | 31/12/2014 |
| Actual date of delivery | | XXX |
| Deliverable number | | 6.2 |
| Deliverable title | | Textual inference components development, III cycle |
| Type | | Program |
| Status and version | | Final |
| Number of pages | | XX |
| Contributing partners | | DFKI, BIU, FBK, UHEI |
| WP leader | | DFKI |
| Task leader | | DFKI |
| Authors | | Kathrin Eichler, Lili Kotlerman, Vivi Nastase, Tae-Gil Noh |
| EC project officer | | Carola Carstens |
| The partners in EXCITEMENT are: | Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Germany | |
| NICE Systems, Israel | |
| Fondazione Bruno Kessler(FBK), Italy | |
| Bar-Ilan University, Israel | |
| Heidelberg University, Germany | |
| OMQ, Germany | |
| ALMAWAVE, Italy | |

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

NICE Systems

EXCITEMENT

Moshe Wasserblat moshew@nice.com

Hapnina 8 Phone: +972 (9) 775-3702

Ra’anana, Israel Fax: +972 (9) 775-3702

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

© 2011, The Individual Authors

No part of this document may be reproduced or transmitted in any form, or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission from the copyright owner.

Table of Contents

[1. Introduction 6](#_Toc360192160)

[1.1 About this Document 6](#_Toc360192161)

[1.2 Introduction to the Transduction Layer 6](#_Toc360192162)

[1.3 Related Terminology 7](#_Toc360192163)

[1.4 Related Documents 7](#_Toc360192164)

[2. Data Flow Overview 8](#_Toc360192165)

[2.1 Decomposition 8](#_Toc360192166)

[2.1.1 Data: Input Data 9](#_Toc360192167)

[2.1.2 Module: Fragment Annotator 9](#_Toc360192168)

[2.1.3 Module: Modifier Annotator 10](#_Toc360192169)

[2.1.4 Module: Fragment Graph Generator 10](#_Toc360192170)

[2.2 Composition Use Case 1 11](#_Toc360192171)

[2.2.1 Module: Graph Merger 12](#_Toc360192172)

[2.2.2 Module: Collapsed Graph Generator 13](#_Toc360192173)

[2.3 Composition Use Case 2 13](#_Toc360192174)

[2.3.1 Module: Node Matcher 14](#_Toc360192175)

[2.3.2 Module: Category Annotator 15](#_Toc360192176)

[3. Core Data Structures 16](#_Toc360192177)

[3.1 Interaction 16](#_Toc360192178)

[3.1.1 class Interaction (eu.excitementproject.tl.structure) 16](#_Toc360192179)

[3.2 Introduction to the Three Graphs 17](#_Toc360192180)

[3.3 Graph Data Structure in Detail 18](#_Toc360192181)

[3.3.1 Fragment Graph 18](#_Toc360192182)

[3.3.2 Raw Graph 23](#_Toc360192183)

[3.3.3 Collapsed Graph 34](#_Toc360192184)

[4. UIMA Type System for Transduction Layer 43](#_Toc360192185)

[4.1 Introduction 43](#_Toc360192186)

[4.2 Types 44](#_Toc360192187)

[4.2.1 Metadata (eu.excitement.type.tl) 44](#_Toc360192188)

[4.2.2 FragmentAnnotation (eu.excitement.type.tl) 44](#_Toc360192189)

[4.2.3 FragmentPart (eu.excitement.type.tl) 45](#_Toc360192190)

[4.2.4 AssumedFragment (eu.excitement.type.tl) 45](#_Toc360192191)

[4.2.5 DeterminedFragment (eu.excitement.type.tl) 46](#_Toc360192192)

[4.2.6 ModifierAnnotation (eu.excitement.type.tl) 46](#_Toc360192193)

[4.2.7 CategoryAnnotation (eu.excitement.type.tl) 47](#_Toc360192194)

[4.2.8 CategoryDecision (eu.excitement.type.tl) 48](#_Toc360192195)

[5. Interface Definitions for the WP6 Modules 49](#_Toc360192196)

[5.1 Interfaces of Decomposition Components 49](#_Toc360192197)

[5.1.1 Fragment Annotator Module: interface *FragmentAnnotator* (eu.excitementproject.tl.decomposition.api) 49](#_Toc360192198)

[5.1.2 Modifier Annotator Module: interface *ModifierAnnotator* (eu.excitementproject.tl.decomposition.api) 50](#_Toc360192199)

[5.1.3 Fragment Graph Generator Module: interface *FragmentGraphGenerator* (eu.excitementproject.tl.decomposition.api) 51](#_Toc360192200)

[5.2 Interfaces of Composition Components 52](#_Toc360192201)

[5.2.1 Graph Merger Module: interface *GraphMerger* (eu.excitementproject.tl.composition.api) 52](#_Toc360192202)

[5.2.2 Collapsed Graph Generator Module: interface *CollapsedGraphGenerator* (eu.excitementproject.tl.composition.api) 53](#_Toc360192203)

[5.2.3 Node Matcher Module: interface *NodeMatcher* (eu.excitementproject.tl.composition.api) 54](#_Toc360192204)

[5.2.4 Category Annotator Module: interface *CategoryAnnotator* (eu.excitementproject.tl.composition.api) 56](#_Toc360192205)

[5.3 Top Level Interface Definition 58](#_Toc360192206)

[5.3.1 Introduction to the Top Level 58](#_Toc360192207)

[5.3.1 Use Case 1 Top Level API: interface *UseCaseOneRunner* (eu.excitementproject.tl.toplevel.api) 59](#_Toc360192208)

[5.3.2 Use Case 2 Top Level API: interface *UseCaseTwoRunner* (eu.excitementproject.tl.toplevel.api) 60](#_Toc360192209)

[6. Implementation of the Prototype 62](#_Toc360192210)

[6.1 Implementation of Decomposition Components 62](#_Toc360192211)

[6.1.1 Fragment Annotator Module: class SentenceAsFragmentAnnotator (eu.excitementproject.tl.decomposition.fragmentannotator) 62](#_Toc360192212)

[6.1.2 Modifier Annotator Module: class AdvAsModifierAnnotator (eu.excitementproject.tl.decomposition.modifierannotator) 62](#_Toc360192213)

[6.1.3 Fragment Graph Generator Module: class FragmentGraphGeneratorFromCAS (eu.excitementproject.tl.decomposition.fragmentgraphgenerator) 63](#_Toc360192214)

[6.2 Implementation of Composition Components 63](#_Toc360192215)

[6.2.1 Graph Merger Module: class AutomateWP2ProcedureGraphMerger (eu.excitementproject.tl.composition.graphmerger) 63](#_Toc360192216)

[6.2.2 Collapsed Graph Generator Module: class SimpleCollapsedGraphGenerator (eu.excitementproject.tl.composition.collapsedgraphgenerator) 64](#_Toc360192217)

[6.2.3 Node Matcher module: class NodeMatcherLongestOnly (eu.excitementproject.tl.composition.nodematcher) 65](#_Toc360192218)

[6.2.4 Category Annotator Module: class CategoryAnnotatorAllCats (eu.excitementproject.tl.composition.categoryannotator) 66](#_Toc360192219)

[6.3 Implementation of Top Levels 66](#_Toc360192220)

[6.3.1 Use Case 1: class UseCaseOneRunnerPrototype (eu.excitementproject.tl.toplevel.usecaseonerunner) 66](#_Toc360192221)

[6.3.2 Use Case 2: class UseCaseTwoRunnerPrototype (eu.excitementproject.tl.toplevel.usecaseonerunner) 68](#_Toc360192222)

[6.4 Implementation of Data Readers, and Other Utilities 68](#_Toc360192223)

[6.4.1 class CASUtils (eu.excitementproject.tl.laputils) 68](#_Toc360192224)

[6.4.2 class InteractionReader (eu.excitementproject.tl.laputils) 69](#_Toc360192225)

[7. Plans for the Next Cycle 70](#_Toc360192226)

[7.1 Provide an Experimentation Environment 70](#_Toc360192227)

[7.2 Provide More Sophisticated Module Implementations 70](#_Toc360192228)

[7.2.1 InteractionReader 70](#_Toc360192229)

[7.2.2 Decomposition 71](#_Toc360192230)

[7.2.3 Composition Use Case 1 71](#_Toc360192231)

[7.2.4 Composition Use Case 2 72](#_Toc360192232)

# Introduction

## About this Document

Deliverable 6.1 is of type “P”, i.e. a program. This document provides a description of this program. The actual source code described in this document has been made available to all project members. A zipped version of the current code and all data necessary to run it has been uploaded to the member area of the project’s website in /Deliverables /Month 18/WP6/source code/. The code can also be found in the Transduction Layer github repository at <https://github.com/hltfbk/Excitement-Transduction-Layer> , which is currently accessible to all WP6 developers and to relevant WP7 developers.

## Introduction to the Transduction Layer

The EXCITEMENT open platform (EOP) developed in WP4 provides the textual entailment capability to decide the entailment relation between pairs of given textual units. This entailment recognition capability itself, however, does not provide a complete solution to the needs of the industrial partners, who aim to use textual entailment for exploring customer interactions (WP7). Additional steps are required to break down the information need of the industrial partners into textual entailment problems and combine the entailment decisions returned by the EOP into a response to the information need. Therefore, we need an additional layer of services on top of the EOP to achieve inference-based exploration of customer interaction data. We call this the *Transduction Layer*.

The Transduction Layer was developed within WP6 for the following two industrial use case scenarios that have been defined within the EXCITEMENT project (see Deliverables 1.1 and 3.1b for more details on the use cases):

*Use Case 1: Entailment graph creation:*

In this use case, the aim is to draw an entailment graph from a set of given interactions.

*Use Case 2: Interaction categorization*

In this use case, the aim is to annotate matching categories on a given interaction, using entailment information.

An analysis of the companies’ use case scenarios has shown that the Transduction Layer can be organized around two central steps, namely *decomposition*, i.e. converting the company’s input into a set of entailment units, from which entailment decision problems can be created, and *composition*, i.e. building entailment pairs and processing the entailment decisions returned by the EOP to meet the company’s information need. The decomposition part is shared by both use cases, the composition part differs.

This document is structured as follows. We first describe the main data flow for the decomposition step (shared by both use cases) and the composition step of use case 1 and use case 2, respectively (chapter 2). We then describe the core data structures (chapter 3) and the UIMA type system (chapter 4) we designed for the Transduction Layer. Chapter 5 contains interface definitions for all transduction layer modules: the interfaces for the core modules and the top level interfaces defined specifically for the industrial partners (WP7). This is followed by a chapter describing the implementation of the prototype, including one prototypical implementation for each defined module (chapter 6). The document ends with a chapter on WP6’s plans for the next cycle (chapter 7).

## Related Terminology

In this document, we will use the following terminology:

* *RTE*: Recognizing textual entailment
* *Use Case 1 / Use Case 2*: This refers to the two use cases introduced in the previous section.
* *EOP*: This refers to the EXCITEMENT open platform providing the RTE functionality.
* *EDA*: This refers to an entailment decision algorithm provided by the EOP, i.e. the part of the open platform that returns an entailment decision for a given text pair.
* *LAP*: This refers to an linguistic analysis pipeline provided by the EOP, i.e. the part of the open platform that creates a JCas object with linguistic annotations.
* *Entailment graph*: An entailment graph orders text units in a structured hierarchy based on the entailment relations that hold between these text units.

## Related Documents

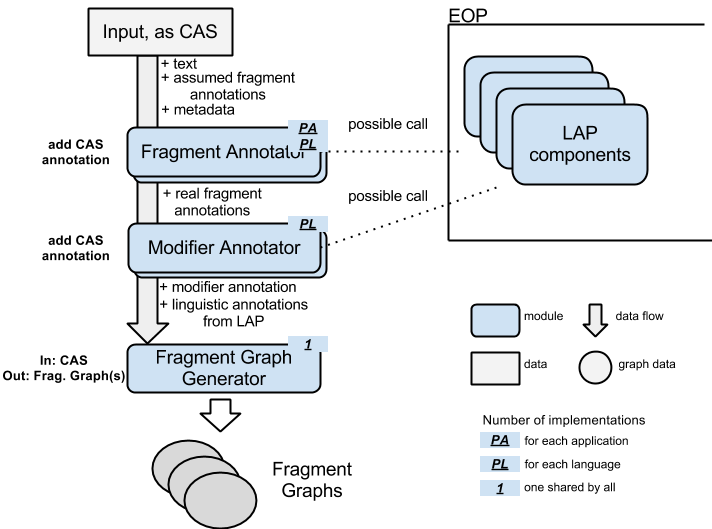
* *Deliverable 1.1*: User Requirements (June 2012)
* *Deliverable 3.1b*: Specification of the Transduction Layer (Sep 2012)
* *EOP specification*: Specifications and architecture for the open platform, I. cycle
* *UIMA Documentation*: UIMA Tutorial and Developers' Guides (<http://uima.apache.org/d/uimaj-2.4.0/tutorials_and_users_guides.html>)

# Data Flow Overview

This section describes the main Transduction Layer data flows. It holds three subsections: [Decomposition](#_Decomposition), [Composition Use Case 1](#_Composition_Use_Case), and [Composition Use Case 2](#_Composition_Use_Case_1).

## Decomposition

The following figure outlines the data flow for the decomposition step and the modules that are part of this data flow. In the decomposition step, the input provided by the user (e.g., WP7) is processed (possibly using LAP components provided by the EOP) and turned into a set of fragments graphs. The decomposition step is relevant to both use cases: In use case 1, it is required because fragment graphs are the data structures, from which entailment graphs are built. In use case 2, it is required because an incoming email is annotated with categories by matching the fragment graphs extracted from the email against an existing entailment graph.



### Data: Input Data

The input from the upper layer – referred to here as “input” – can be one customer interaction (in use cases 1 and 2), or a category description (in use case 2). We make the assumption that this input consists of *text* and *assumed fragment* annotations. *text* is the actual text of the interaction or category description. The *assumed fragments* are annotations that span portions of the *text,* which are considered to express relevant content. One input can have any number of assumed fragment annotations, including none.

Within the Transduction Layer, the input is represented as a UIMA CAS. The *text* is given in the CAS's Sofa (Subject of Analysis), and each *assumed fragment* is given as a CAS annotation on the text. A specific type is used for this annotation.

The caller (WP7) can directly prepare this CAS. WP6 provides some helper functions that enable users to annotate fragments and modifiers (for details, refer to the following sections) without understanding the internals of CAS. If the WP7 input does not need to mark annotations (fragments or modifiers), the set of interactions can be passed in a simpler data type (String based List<Interaction>). For the actual interfaces, please see section 5.3.1.

### Module: Fragment Annotator

A Fragment Annotator is a module that generates *determined fragment annotations*. By “determined” we mean fragment annotations determined by this module that are used in later TL steps.

There are several reasons for performing this additional fragment annotation step: (i) there are no fragment annotations provided by the user; (ii) the fragment annotations provided by the user are too broad, covering coordinate, subordinate or complex clauses (e.g. “The food was bad and the leg room was too small”). If no fragment annotations are provided, the module performs its own analysis of the text, and produces *determined fragment annotations*. If *assumed fragments* were given, the module iterates over them, and refines them if they are found to be complex expressions to produce the *determined fragment annotations*. The span of the determined fragment annotations may coincide with a fragment annotation provided by the user, or can cover contiguous or non-contiguous portions of the user's annotation.

The annotations produced are added to the input CAS, enriching the text's representation.

This module is application- and language-specific.

The module may (need to) call an LAP, depending on the implementation. If it calls LAP, it must consider future steps and try to minimize the need of future LAP calling.

### Module: Modifier Annotator

After obtaining fragment annotations, the next step is to identify all modifiers within these fragments. The identified modifiers are annotated with a specific *modifier annotation type.* The words in a fragment that are not annotated as *modifier*, form the *base statement* (also called *base predicate* in WP2 terms). We simply keep one modifier annotation type, but no base statement annotation (non-modifier) type.

An implementation of this module marks all modifiers in the fragments. The module adds annotations to the given CAS, and does not generate any independent data.

We expect the modifier identification to be language specific, and thus language specific implementations of this module to be necessary.

The module may (need to) call an LAP, since detecting modifiers (probably) needs information of POS tags or more. When it calls LAP, it must consider future steps and try to minimize the need of future LAP calling.

### Module: Fragment Graph Generator

This module consumes one CAS, and generates one or more *fragment graphs* (one fragment graph for each determined fragment).

The input CAS of this module has the following annotations at this stage.

* *[Group A]* Determined fragment annotation, modifier annotation
* *[Group B]* Linguistic annotations from LAP, Metadata from the user
* *[Others]* Assumed fragment annotation from the user

Note that the input CAS holds metadata from the interaction XML. This includes language, channel, provider, date, category, etc. To see the full list of metadata, please check the Metadata type definition (section 4.2.1). Please note that the language of the CAS is a special metadata and stored in the CAS itself. It can be accessed using a*JCas.setDocumentLanguage()* and *aJCas.getDocumentLanguage()*, respectively.

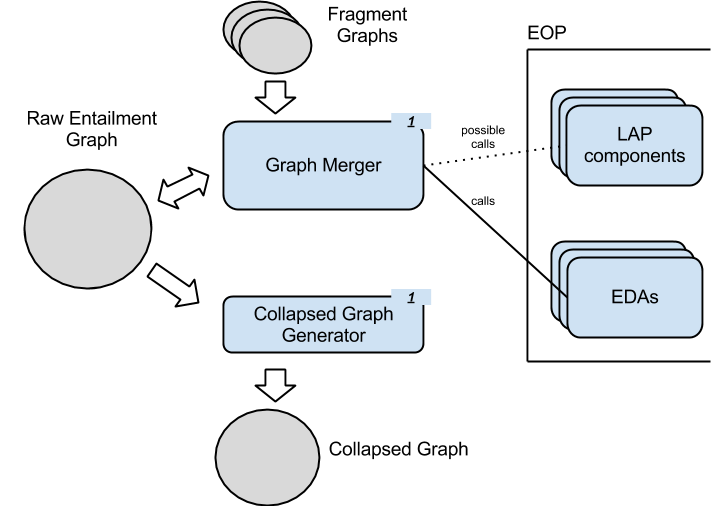
The fragment graphs are built based on the fragment and modifier annotations (group A). The fragment graph corresponding to a fragment is built by producing as nodes all combinations of base statement and modifiers, with entailment relation between nodes based on subsumption of the corresponding sets of covered modifiers. The annotations from group B are used to provide additional information that is stored in each node of a fragment graph, to be available in successive annotation steps. Other annotations (like assumed fragmentation) are not used.

Each fragment graph is represented as a specifically designed Java object. This representation is detailed in section 3.3.1. As the input CAS may contain more than one determined fragment annotations, the output of this module is a set of fragment graphs (one per determined fragment).

When the CAS is consumed and the associated fragment graph(s) are built, one cycle of the decomposition flow is finished.

## Composition Use Case 1

The following figure outlines the data flow of the composition step for use case 1 (entailment graph building) and the modules that work for this data flow. In this composition step, the fragment graphs created from the input data in the decomposition step are merged into an entailment graph (possibly calling LAP components and EDAs provided by the EOP) and collapsed to the final output (a collapsed graph).



### Module: Graph Merger

This module builds or grows a raw entailment graph (also referred to as *raw graph*), by merging fragment graphs. It receives as input a raw graph (possibly empty), and a set of fragment graphs that are gradually added to the input raw graph. The result of this processing is a bigger, richer, version of the input raw entailment graph. For one instance of an industrial application there exists only one raw entailment graph, which grows with each run of the Graph Merger module.

Each raw graph is represented through a specially designed Java object. For this, see section 3.3.2.

To merge fragment graphs into the raw graph, the Graph Merger module uses the entailment decision capability of the EXCITEMENT open platform (EOP). For more information about the EOP, please see the EOP specification.

We expect this module to be application-independent. This means that the unique module implementation must provide a sound way of choosing the most fitting entailment decision algorithm (EDA) from the EOP.

To produce input data for the chosen EDA in an efficient manner (avoiding unnecessary LAP calls and creation of CAS objects), this module should try to reuse as much of the LAP annotations already attached to the nodes of the raw graph as possible.

### Module: Collapsed Graph Generator

This module trims an input raw graph (by selecting edges, creating equivalence classes, etc.) in order to produce a special version of the entailment graph – we call it *collapsed* graph – that is useful for the application scenario.

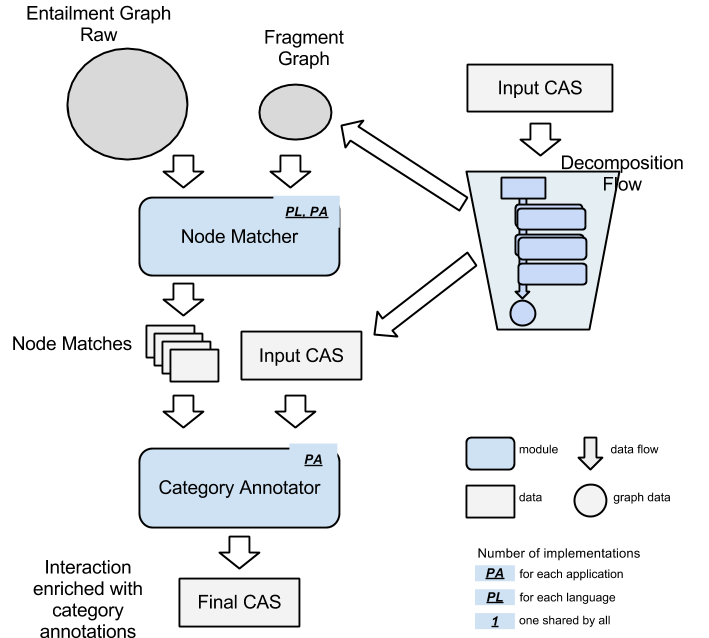
The raw graph essentially represents all Transduction Layer’s knowledge about the entailment relations between its nodes. The purpose of the collapse procedure is to make final decisions on whether an entailment relation holds or not between the nodes, resolve transitivity violations and compress paraphrasing statements into equivalence class nodes.

Each collapsed graph is represented through a specially designed Java object. For this, see section 3.3.3.

The module is self-contained – it transforms the input raw graph into a collapsed graph without relying on external modules (such as the EOP) or data. A confidence threshold may be provided as an additional input parameter (e.g., by an industrial system), to customize the resulting collapsed graph by filtering entailment relations from the input raw graph based on their strength.

## Composition Use Case 2

The following figure outlines the data flow of the composition step for use case 2 (category annotation), and the modules used for this data flow. In this composition step, the fragment graphs created from an incoming email are matched against an existing entailment graph. Extracting and combining category information from the matches, the incoming email is then enriched with matching categories and associated confidence scores.



### Module: Node Matcher

This module matches a given fragment graph *F* against a given raw entailment graph *R*. It returns a set of *node matches*, where each node match holds a node *M* (one of the nodes in *F*) associated to a set of *per node scores*. Each per node score is a tuple <*E*,*p*>, where *E* denotes a node in *R* and *p* denotes the confidence of *M* matching *E*. Node matches and per node scores are represented through specially designed Java objects. For details, refer to sections 5.2.3.3.1 and 5.2.3.3.2

The module aims for a fast (search-engine like) matching to produce results in near-real-time.

We expect this module to be language- and application-specific, thus several implementations may be necessary. It does not need calls on external modules (like LAP or EOP), or stored data, other than the fragment graph and the raw graph itself.

### Module: Category Annotator

This module adds category annotation to a given input CAS. In addition to the input CAS, it takes as input the output of the Node Matcher module, i.e. a set of node matches for a particular fragment. It extracts category information from those nodes, and uses this information to compute confidence scores for all categories retrieved in the node matches. It then adds this category confidence as new annotation to the input CAS.

This module is application-specific.

There are no external dependencies expected for this module.

# Core Data Structures

This chapter introduces the core data structures used in the Transduction Layer (TL): First, the data structure *Interaction*, which holds the input provided by the user, and, second, the graph data structures required for building entailment graphs.

## Interaction

class Interaction (eu.excitementproject.tl.structure)

This section describes the Interaction class, which represents one un-annotated interaction text and its metadata. Note that this data structure is a "boundary" data structure that is designed to get external input and translate it to the input CAS data type. The data type itself is not the main target of processing: actual processing like annotations and fragment graph building is always happening on the input CAS level.

* Attributes:
  + interactionString: holds the interaction text itself as one string
  + lang: metadata language ID
  + interactionID: the id of the interaction
  + channel: metadata channel
  + provider: metadata provider
  + category: metadata category
* Constructors:

*Interaction(String interactionString, String langID, String interactionId, String category, String channel, String provider)*

This is the full constructor that gets everything. You can set null on metadata. But you can never set null on *interactionString* and *langID*. (Minimally, those two are needed to construct an input CAS.)

*Interaction(String interactionString, String langID)*

This is the minimal constructor that sets only interaction string and language ID. This is the absolute minimum.

*Interaction(String interactionString, String langId, String interactionId, String category)*

Another constructor with some more parameters.

* Methods:

*void fillInputCAS(JCas aJCas)*

This method gets one JCas for the TL layer, and fills it with the interaction. It sets interaction string, language ID, and metadata.

* + - @param aJCas: a JCas

*JCas createAndFillInputCAS()*

This method first generates a new JCas, and fills it calling fillInputCas()

* + - @return (JCas): the resulting JCas

(The usual get methods for each attribute.)

## Introduction to the Three Graphs

As we have seen in section 3, we have three conceptually different, graph-based data representations. One is *fragment graph*, which is a graph built from a fragmentation by identifying modifiers in the fragment. Another graph is the raw entailment graph (or *raw graph)*: this is the main entailment graph that is being kept and worked with in WP6. Major operations like adding edges and required EDA calling (for entailment judgment) is all done with this graph. Finally, the last graph is the so-called *collapsed graph* (the trimmed graph). This graph can be automatically generated from the raw graph (via a module).

To implement the graph structures we use the JGraphT library, which provides a rich and flexible inventory of graph types, as well as visualization functionalities. The JGraphT library (<https://github.com/jgrapht/jgrapht>) offers implementations for directed and undirected, weighted and unweighted simple- and multi-graphs.

## Graph Data Structure in Detail

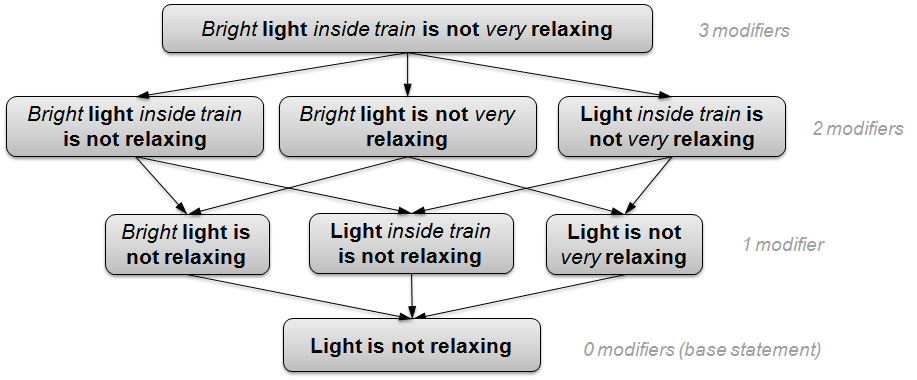
The choice for representing the three graph types as described in the following sections was driven by the structural and functional requirements for each of them.

### Fragment Graph

The fragment graphs, with their corresponding class *FragmentGraph*, are simple graph structures. Their nodes contain much information, but structurally they are simple directed graphs. A directed edge – representing the entailment relation – connects the node corresponding to a text fragment T with a node corresponding to the text fragment T minus one of T’s modifiers.

We assume a text fragment to be composed of a base statement (BS) plus a number of modifiers (M). A node of this graph corresponds to BS + M1 ... Mk. We assume a textual entailment (TE) relation (i.e. an edge in the graph) between every two statements (Si, Sj) that differ only by one modifier: Si = Sj + Mx => Si *entails* Sj.

An example of a fragment graph is presented below. Nodes hold text fragments with modifiers shown in italics.



#### class FragmentGraph (eu.excitementproject.tl.structures.fragmentgraph)

This class extends the DefaultDirectedWeightedGraph class, because the graph is directed and we might decide to have the edges weighted. Currently they are not. Please refer to the JavaDoc for the DefaultDirectedWeightedGraph class for information about inherited methods: <http://jgrapht.org/javadoc/org/jgrapht/graph/DefaultDirectedWeightedGraph.html>

The nodes of the *FragmentGraph* are *EntailmentUnitMention*-s, and the edges are *FragmentGraphEdge*-s.

* Attributes:
  + *EntailmentUnitMention baseStatement* – the text fragment for which the fragment graph was built, without any modifiers.
  + *EntailmentUnitMention topStatement* – the (complete) text fragment for which the fragment graph was built.
  + *org.apache.uima.jcas.JCas document* – the CAS object with the annotation layers described in Section [UIMA Type System for Transduction Layer](#_UIMA_type_system)
  + *FragmentAnnotation fragment* – the (determined) fragment annotation from the input CAS for which the fragment graph was built.
  + *int depth* – the depth of the fragment graph, equivalent to the number of modifiers in the corresponding text fragment.
* Constructors:

The constructors build a fragment graph based on a JCas with fragment and modifier annotations. It generates one node for each combination of “base statement” and *k* modifiers (*k* ranges between 0 and the total number of modifiers in the given text), and connects them based on the modifier set subsumption.

*FragmentGraph(org.apache.uima.jcas.JCas aJCas, FragmentAnnotation frag)*

Builds a fragment graph from a (determined) fragment in a CAS object corresponding to a document based on the modifier annotations in the fragment.

* @param aJCas – the CAS object corresponding to a document
* @param frag – a (determined) fragment annotation in the CAS
* Methods:

*buildGraph(JCas aJCas, FragmentAnnotation frag, Set<ModifierAnnotation> modifiers, EntailmentUnitMention parent)*

Builds a fragment graph based on a CAS; starts with the top node that has all modifiers, removes them one by one and recursively builds the graph.

* + - @param aJCas – document CAS object
    - @param frag – (determined) fragment
    - @param mods – set of modifiers
    - @param parent – parent node (that has one extra modifier compared to the current node)

*boolean consistentModifiers(Set<ModifierAnnotation> sma)*

Checks if a set of modifiers is consistent, i.e. – it doesn't miss a modifier that another depends on. Example:

Seats are uncomfortable as too old.

=> Seats are uncomfortable as old (OK)

=> Seats are uncomfortable as too (not OK)

* + - @param sma – a set of modifier annotations from the document CAS
    - @return (boolean) – true if the set of modifiers given is consistent

*EntailmentUnitMention getVertex(EntailmentUnitMention eum)*

* + - @param eum – an entailment unit mention
    - @return (EntailmentUnitMention) – the node in this fragment graph that equals the given entailment unit mention, or the given entailment unit mention if no equal node exists.

*EntailmentUnitMention getBaseStatement()*

* + - @return (EntailmentUnitMention) – the graph node that corresponds to the base statement of the graph

*EntailmentUnitMention getCompleteStatement()*

* + - @return (EntailmentUnitMention) – the graph node that corresponds to the complete (top) statement of the graph

*Set<ModifierAnnotation>getFragmentModifiers(org.apache.uima.jcasJCas aJCas, FragmentAnnotation f)*

* + - @param aJCas – the CAS object corresponding to the document
    - @param f – the fragment annotation for the fragment corresponding to *this* fragment graph
    - @return (Set<ModifierAnnotation>) – the set of modifier annotations contained in the given fragment

*int getMaxLevel()*

* + - @return (int) – the depth of *this* fragment graph

*Set<EntailmentUnitMention> getNodes(int level)*

* + - @param level – a level in the graph
    - @return (Set<EntailmentUnitMention) – the nodes in the graph that have *level* number of modifiers

*EntailmentUnitMention addNode(EntailmentUnitMention eum)*

* + - @param eum – the node to be added to the graph
    - @return (EntailmentUnitMention) – the node added

Methods for internal testing purposes

*getSampleGraph(), getSampleOutput()*

Specialized implementations of general graph methods

*addEdge(EntailmentUnitMention parent, EntailmentUnitMention eum), containsVertex(EntailmentUnitMention eum), toString()*

#### class EntailmentUnitMention (eu.excitementproject.tl.structures.fragmentgraph)

An *EntailmentUnitMention* refers to a piece of text (fragment or subfragment in WP2 terminology) occurring within a particular input text, i.e., it is associated to exactly one document. *EntailmentUnitMention*-s referring to the same text are grouped within a single *EntailmentUnit* (for details, see section 3.3.2.2). A vertex in the fragment graph is an EntailmentUnitMention. It consists of a base statement + a number of modifiers.

* Attributes:
  + *String text* – the text fragment corresponding to the node
  + *int begin* – starting position of the fragment relative to the document it comes from
  + *int end* – end position of the fragment relative to the document it comes from
  + *String categoryId* – category information
  + *int level* – the level of the node in the fragment graph it is part of. It corresponds to the number of modifiers the text has
  + *Set<ModifierAnnotation>modifiers* – the set of modifiers (represented as the set of corresponding annotations from the CAS object) contained in this node’s text fragment
  + *Set<SimpleModifier>modifiersText* – the set of modifiers in the text, represented through SimpleModifier objects (modifier text (String) and position of the modifier – begin (int), end (int) – relative to the text fragment)
* Constructors:

The constructors build an EntailmentUnitMention object for a given text fragment, such that it contains only the specified modifiers

*EntailmentUnitMention(org.apache.uima.jcas.JCas aJCas, FragmentAnnotation frag, java.util.Set<ModifierAnnotation> mods)*

* + - @param aJCas – the document CAS object
    - @param frag – a fragment annotation in the CAS
    - @param mods – a set of modifiers (represented as a set of modifier annotations from the CAS)
    - builds an EntailmentUnitMention based on the (determined) fragment annotation in a document CAS object, that covers the given set of modifiers (and only those)

*EntailmentUnitMention(java.lang.String textFragment)*

Builds an EntailmentUnitMention for the given text portion, with no modifiers

* + - @param textFragment – a text fragment
* Methods:

*int getBegin()*

* + - @return (int) – the start position of the node’s text fragment (relative to the document the fragment belongs to)

*int getEnd()*

* + - @return (int) – the end position of the node’s text fragment (relative to the document the fragment belongs to)

*int getLevel()*

* + - @return (int) – the level of the node in the graph (i.e. how many modifiers it covers)

*String getCategoryId()*

* + - @return (String) – the category metadata of the document this node originates from

*Set<ModifierAnnotation>getModifierAnnotations()*

* + - @return (Set<ModifierAnnotation>) – the set of modifier annotations (from the CAS)

*Set<SimpleModifier>getModifiers()*

* + - @return (Set<SimpleModifier>) – modifiers information in the SimpleModifier format (text and position)

*String getText()*

* + - @return (String) – the node’s text

Specialized implementations of generic Object methods

*equals(EntailmentUnitMention eum), toString()*

#### class FragmentGraphEdge (eu.excitementproject.tl.structures.fragmentgraph)

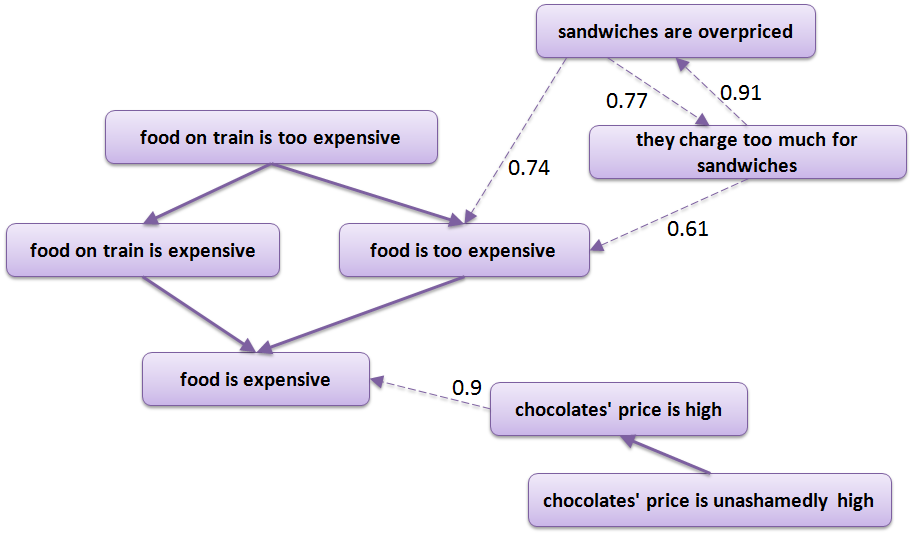
This is the edge class for the FragmentGraph. For now, edges are directed, with default weight. This class extends DefaultEdge, documented here: <http://jgrapht.org/javadoc/org/jgrapht/graph/DefaultEdge.html>

This class has the same attributes and methods as the class DefaultEdge: *source*, *target*, *weight* attributes; methods to obtain the source, target and weight information, and the expected constructor.

### Raw Graph

The raw graphs, with their corresponding class *EntailmentGraphRaw*, are directed multigraphs (have multiple directed edges between the same pair of nodes) obtained by merging fragment graphs. The choice to represent this structure as a multigraph is motivated by the possibility that the entailment decision between a pair of nodes could be obtained from different entailment decision algorithms (EDAs). The different edges are combined when building a collapsed entailment graph, after all fragments graphs were merged into one raw graph.

An example of a raw graph is presented below. Dashed edges represent EDA entailment decisions with confidence scores, while solid edges correspond to edges copied from fragment graphs. For clarity reasons, in this example we do not show edges that correspond to “no entailment” decisions of the EDA.



The nodes of a raw graph are entailment units (*EntailmentUnit*), which cover a set of entailment unit mentions that express the same text fragment.

The edges of a raw graph are entailment relations (*EntailmentRelation*), which hold the information about whether the entailment relation holds between the edge’s source and target nodes.

#### class EntailmentGraphRaw (eu.excitementproject.tl.structures.rawgraph)

This class contains the graph structure for the raw graph. We call it *EntailmentGraphRaw*. This graph grows by adding to it *FragmentGraph*-s by "merging", which is done through the [Graph Merger](#_Graph_merger_module:) interface. The nodes are entailment units (*EntailmentUnit*), and the edges (*EntailmentRelation*) are generated based on decisions from the EDA. As such there can be several edges between the same two nodes, each corresponding to one EDA result. Edges can also be added based on graph structure or prior knowledge (e.g.by copying edges contained in fragment graphs). This graph extends DirectedMultigraph, to allow for multiple directed edges between the same two nodes. The JavaDoc for the DirectedMultigraph for information about inherited methods can be found here: <http://jgrapht.org/javadoc/org/jgrapht/graph/DirectedMultigraph.html>

* Attributes:

This graph has no particular attributes, other than those inherited from its parent class (MultiGraph).

* Constructors:

*EntailmentGraphRaw(FragmentGraph fg)*

The constructor builds a raw graph from a given FragmentGraph

* + - @param fg – the fragment graph

*EntailmentGraphRaw(java.io.File xmlFile)*

The constructor uploads an existing raw graph from the given xml file. The file is assumed to follow the format of the output produced by toXML() method of the EntailmentGraphRaw class.

* + - @param xmlFile – the xml file from which to load a previously produced raw entailment graph.
    - throws EntailmentGraphRawException if creating the graph from the given file did not succeed for some reason (file not found, wrong xml format etc.)

*EntailmentGraphRaw()*

The constructor generates an empty raw entailment graph.

* Methods:

Methods for building the graph:

*EntailmentRelation addEdgeFromEDA(EntailmentUnit sourceVertex,  
 EntailmentUnit targetVertex,  
 eu.excitementproject.eop.common.EDABasic<?> eda)*

The method calls the specified EDA in order to obtain the entailment decision and adds the corresponding edge from the given source vertex to the given target vertex.

* + - @param sourceVertex – the source vertex (must be contained in the graph)
    - @param targetVertex – the target vertex (must be contained in the graph)
    - @param eda – an instance of the EDA that should be used to calculate the entailment decision.
    - @return (EntailmentRelation) – the method returns the edge, which was added to the graph. If no edge was added because the graph already contained such an edge, the method returns *null*.
    - @throws EntailmentGraphRawException if the edge could not be added for some reason (e.g. if the source or target node was not found in the graph)

*EntailmentRelation addEdgeFromFragmentGraph(FragmentGraphEdge fragmentGraphEdge, FragmentGraph fg)*

The method copies the specified edge from the given fragment graph to the raw graph.

* + - @param fragmentGraphEdge – the fragment graph edge to be copied. If the source and the target nodes of the edge are not found in the raw graph, they will be created prior to adding the edge. If the nodes are already found in the graph, their attributes are updated.
    - @param fg – the fragment graph from which to copy the edge.
    - @return (EntailmentRelation) – the method returns the edge, which was added to the graph. If the edge already existed in the graph, the method returns null.

*EntailmentRelation addEdgeByInduction(EntailmentUnit sourceVertex,  
 EntailmentUnit targetVertex,  
 java.lang.Double confidence)*

The method adds an entailment edge with the given confidence from the given source vertex to the given target vertex. The method should be used when an edge is induced (with some confidence) based on the knowledge of the graph structure. Currently this method is used by the [AutomateWP2ProcedureGraphMerger](#_Graph_Merger_module:_1), which adds *entailment* edges for upper-level nodes (i.e. nodes with more than 1 modifier) as soon as there is entailment between specific nodes on lower levels.

* + - @param sourceVertex – the source vertex (must be contained in the graph)
    - @param targetVertex – the target vertex (must be contained in the graph)
    - @param confidence – confidence that entailment indeed holds between the two nodes in the given direction.
    - @return (EntailmentRelation) – the edge, which was added to the graph. If the edge already existed in the graph, the method returns null.
    - @throws EntailmentGraphRawException if the edge could not be added for some reason (e.g. if the source or target node was not found in the graph)

Methods for traversing and querying the graph:

*EntailmentUnit getVertex(java.lang.String text)*

* + - @return (EntailmentUnit) – the node, which corresponds to the given text. If such a node is not found in the graph, returns null.
    - @param text – the text of the node to be returned

*Set<EntailmentUnit> getBaseStatements()*

* + - @return (Set<EntailmentUnit>) – the sets of nodes, which represent base statements, i.e. statements with no modifiers.

*Set<EntailmentUnit> getAllNodes(EntailmentUnit sourceNode, EntailmentUnit targetNode, Set<EntailmentUnit> nodesToReturn)*

* + - @return (Set<EntailmentUnit>) – the set of all nodes, which form the possible paths from sourceNode to targetNode (including the sourceNode and the targetNode). If there is no path between the two nodes, it returns an empty set. If targetNode is the same as sourceNode, the method returns a set with a single EntailmentUnit. The method recursively updates the set *nodesToReturn*, which it obtains as its parameter.
    - @param sourceNode – the source node (must be contained in the graph)
    - @param targetNode – the target node (must be contained in the graph)
    - @param nodesToReturn – the set of nodes, which is recursively updated by the method. Needs to be null or an empty set.
    - @throws EntailmentGraphRawException if the required set of nodes could not be created by some reason (e.g. if source or target node was not found in the graph)

*Hashtable<Integer, Set<EntailmentUnit>>  
 getFragmentGraphNodes(EntailmentUnit baseStatementNode, String completeStatementText)*

* + - @return (Hashtable<Integer, Set<EntailmentUnit>>) – reconstructs and returns the nodes of the fragment graph defined by the given base statement and the given complete statement. The nodes are returned in sets corresponding to different statement levels (numbers of modifiers), which are the keys of the Hashtable.
    - @param baseStatementNode – the node of the raw graph, which holds the base statement (must be contained in the graph).
    - @param completeStatementText – the text of the complete statement, which defines the fragment graph (the same base statement might be produced from different complete statements).
    - @throws EntailmentGraphRawException if the required output could not be produced by some reason (e.g. if the base statement node was not found in the graph, or the given base statement is not related to the given complete statement, etc.)

*Set<EntailmentUnit> getEntailingNodes(EntailmentUnit node)*

*Set<EntailmentUnit> getEntailingNodes(EntailmentUnit node, int level)*

* + - @return (Set<EntailmentUnit>) – the methods return the nodes, which directly entail the given node.
    - @param node – the node whose entailing nodes should be returned
    - @param level – if specified, retains in the output only the nodes with the corresponding value of level (number of modifiers)

*Set<EntailmentUnit> getEntailedNodes(EntailmentUnit node, boolean isSameGraph)*

*Set<EntailmentUnit> getEntailedNodes(EntailmentUnit node, int level, boolean isSameGraph)*

* + - @return (Set<EntailmentUnit>) – the methods return the nodes that are directly entailed by the given node.
    - @param node – the node whose entailed nodes should be returned
    - @param level – if specified, retains in the output only the nodes with the corresponding value of level (number of modifiers)
    - @param isSameGraph – if true, the method will return only the nodes, which belong to the same fragment graph as the input node.

*boolean isEntailment(EntailmentUnit entailingNode, EntailmentUnit entailedNode)*

* + - @return (boolean) – the methods returns *true* if there is an edge entailingNode -> entailedNode. If such edge is not found, it returns *false*.
    - @param entailingNode – the candidate entailing node
    - @param entailedNode – the candidate entailed node

*boolean isEntailmentInAnyDirection(EntailmentUnit nodeA, EntailmentUnit nodeB)*

* + - @return (boolean) – the methods returns *true* if there is an edge connecting the two given nodes in any direction (nodeA-> nodeB or nodeB -> nodeA). If no such edge is found, it returns *false*.
    - @param nodeA
    - @param nodeB

Methods to output the graph:

*void toXML(java.lang.String filename)*

The method saves the graph to an xml file with the given filename.

* + - @param filename – the name of the xml file to which the graph should be saved.
    - throws EntailmentGraphRawException if saving the graph to the given file did not succeed for some reason.

*java.lang.String toDOT()*

*void toDOT(java.lang.String filename)*

The method generates a single string, which contains the graph in DOT format (http://en.wikipedia.org/wiki/DOT\_(graph\_description\_language)) for visualization. The string is either returned or saved to the given output file (if provided).

* + - @return (*java.lang.String*) – the string with the graph in DOT format.
    - @param filename – the name of the output file. If provided, the string will be saved to the file rather than returned.
    - throws IOException if saving to the file failed for some reason.

Methods for internal testing purposes

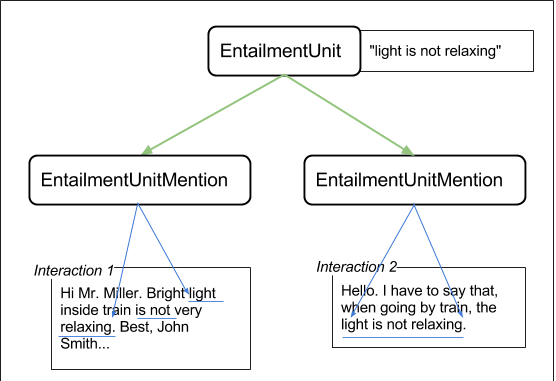
*getSampleOutput(boolean randomEdges)*

Specialized implementations of general graph methods

*toString*()

#### class EntailmentUnit (eu.excitementproject.tl.structures.rawgraph)

An *EntailmentUnit* refers to a piece of text that can occur in one or more input texts. *EntailmentUnit*-s form the nodes of the raw entailment graph. Each such node covers *EntailmentUnitMention*-s that represent the same text. The relationship between entailment unit mentions and entailment units is illustrated in the following figure.



*EntailmentUnit* texts are unique, i.e. two different *EntailmentUnit* objects within the same raw graph should never hold the same text.

* Attributes:
  + *String text* – the canonical text of the entailment unit
  + *Set<EntailmentUnitMention> mentions* – set of all the EntailmentUnitMention-s, represented by the canonical text.
  + *int level* – the number of modifiers in the canonical text fragment. For base statements, level = 0. Negative value stands for “unknown”.
  + *Set< java.lang String> completeStatementTexts* – set of the complete statements, from which the current entailment unit was derived. Since the same node (the same canonical text) can be part of several fragment graphs, we need to keep track of each node’s origins.
  + *Set< java.lang String> interactionIds* – set of the ids of all the interactions, in which the entailment unit was mentioned.
* Constructors:

*EntailmentUnit(EntailmentUnitMention eum, java.lang.String completeStatementText)*

This constructor creates an entailment unit based on an entailment unit mention and a complete statement, in which this mention occurred. This constructor is used to create raw graph nodes from fragment graph nodes.

* + - @param eum – the entailment unit mention
    - @param completeStatementText – the text of the complete statement.

*EntailmentUnit(java.lang.String textFragment, int level, java.lang.String completeStatementText)*

This constructor creates an entailment unit given a text, its number of modifiers (level) and its corresponding complete statement. This constructor is currently used to support the internal testing purposes, namely the raw graph’s getSampleOutput() method.

* + - @param text – the canonical text of the entailment unit
    - @param level – the number of modifiers in the text
    - @param completeStatementText – the text of the complete statement.

*EntailmentUnit(java.lang String text, Set< java.lang String> completeStatementTexts, Set<EntailmentUnitMention> mentions, Set< java.lang String> interactionIds, int level) {*

This constructor receives all the attributes of an entailment unit and generates the entailment unit correspondingly. It is used when loading a previously saved graph from an xml file.

* + - @param text
    - @param completeStatementTexts
    - @param mentions
    - @param interactionIds
    - @param level
* Methods:

*String getText()*

* + - @return (java.lang.String)

*Set<EntailmentUnitMention> getMentions()*

* + - @return (Set<EntailmentUnitMention>)

*Set<String> getMentionTexts()*

* + - @return (Set<java.lang.String>)

*void addMention(EntailmentUnitMention n)*

* + - @param n – the entailment unit mention, which is added to the set of mentions.

*int getNumberOfCompleteStatements()*

The method returns the size of the set of completeStatementTexts, i.e. the number of fragment graphs, in which the entailmet unit was seen.

* + - @return the number of fragment graphs, in which the entailmet unit was seen.

*Set<java.lang.String> getCompleteStatementTexts()*

* + - *@*return (Set<java.lang.String>)

*Set<java.lang.String> getInteractionIds()*

* + - *@*return (Set<java.lang.String>)

*void addCompleteStatement(java.lang.String completeStatementText)*

* + - @param completeStatementText – the text of a complete statement, which is added to the set of complete statements. If the set already contains such complete statement, it is not added.

*int getLevel()*

* + - @return (int)

boolean *isBaseStatement()*

* + - @return (boolean) – returns *true* if level=0, otherwise returns *false*.

Specialized implementations of general Object methods

*equals()* returns *true* if two entailment units have the same canonical text,

*hashCode(), toString()*

#### class EntailmentRelation (eu.excitementproject.tl.structures.rawgraph)

This is the edge type for the raw graph (*EntailmentGraphRaw*). The edge "value" is a textual entailment decision (*eu.excitementproject.eop.common.TEDecision*) obtained from the EDA. The TEDecision object also stores (among other things) a decision label (*eu.excitementproject.eop.common.DecisionLabel*). For details on the *TEDecision* and *DecisionLabel* data types, see the EOP specification. The class extends DefaultEdge: <http://jgrapht.org/javadoc/org/jgrapht/graph/DefaultEdge.html>

* Attributes:
  + *EntailmentUnit source* – the source (entailing) node of the edge
  + *EntailmentUnit target* – the target (entailed) node of the edge
  + *eu.excitementproject.eop.common.TEDecision edge* – TEDecision on the entailment relation source -> target
  + *EdgeType edgeType* – one of the values from the EdgeType enum, namely *EDA, FRAGMENT\_GRAPH, INDUCED, UNKNOWN*, reflecting the type of knowledge that allowed to create this edge.
  + *EDABasic<?> eda* – for *EDA* edge type, this attribute holds the EDA, which produced the edge.
  + *LAPAccess lap* – for EDA edge type, this attribute holds the LAP required by the corresponding EDA.
* Constructors:

*EntailmentRelation(EntailmentUnit source, EntailmentUnit target, eu.excitementproject.eop.common.EDABasic<?> eda)*

This constructor uses the given EDA to compute an entailment decision for *source -> target* and creates the corresponding *edge* attribute.

* + - @param source – the source (entailing) node of the edge
    - @param target – the target (entailed) node of the edge
    - @param eda – an instance of the EDA, which should be used to calculate the entailment decision.
    - @param lap – an instance of the LAP, which should be used by the EDA to calculate the entailment decision.

*EntailmentRelation(EntailmentUnit source, EntailmentUnit target, eu.excitementproject.eop.common.TEDecision edge, EdgeType edgeType)*

This constructor creates an edge without calling an EDA. It is used when copying edges from fragment graphs and adding edges induced from the graph’s structure, as well as when loading a previously saved graph from an xml file.

* + - @param source – the source (entailing) node of the edge
    - @param target – the target (entailed) node of the edge
    - @param edge – TEDecision for the edge
    - @edgeType – the type of the edge
* Methods:

*TEDecision computeTEdecision()*

* + - @return (*eu.excitementproject.eop.common.TEdecision*) – returns the entailment decision computed for the *source* and *target* attributes (*source -> target*) using the entailment unit’s *eda* attribute.

*JCas generateTHPairCAS()*

* + - @return (JCas) – returns a JCAS with data prepared for calling the EDA in order to obtain the entailment decision for *source -> target*

*double getConfidence()*

* + - @return (double) – returns the confidence, stored within the *edge* attribute, that the specified entailment decision is correct.

*EDABasic<?> getEda()*

* + - @return (EDABasic<?>) – returns the entailment relation’s EDA

*LAPAccess getLap()*

* + - @return (*LAPAccess*) – returns the entailment relation’s LAP

*DecisionLabel getLabel()*

* + - @return (eu.excitementproject.eop.common.DecisionLabel) – returns the decision label, stored within the *edge* attribute

*TEDecision getTEdecision()*

* + - @return (*eu.excitementproject.eop.common.*TEDecision) – returns the entailment decision (the *edge* attribute)

Specialized implementation of graph edge methods

*getSource(), getTarget(), toString()*

### Collapsed Graph

A collapsed graph, implemented in class *EntailmentGraphCollapsed*, is a simple directed graph, obtained from the raw graph by collapsing nodes corresponding to equivalent text fragments (from the point of view of the entailment relation), and multiple edges between the same pair of nodes, as well as resolving conflicts (transitivity violations) resulting from automatic entailment decisions.

By definition, collapsed graph is a transitive graph.

The nodes of the collapsed graph are equivalence classes (*EquivalenceClass*), which cover a set of entailment units which cover semantically equivalent text fragments.

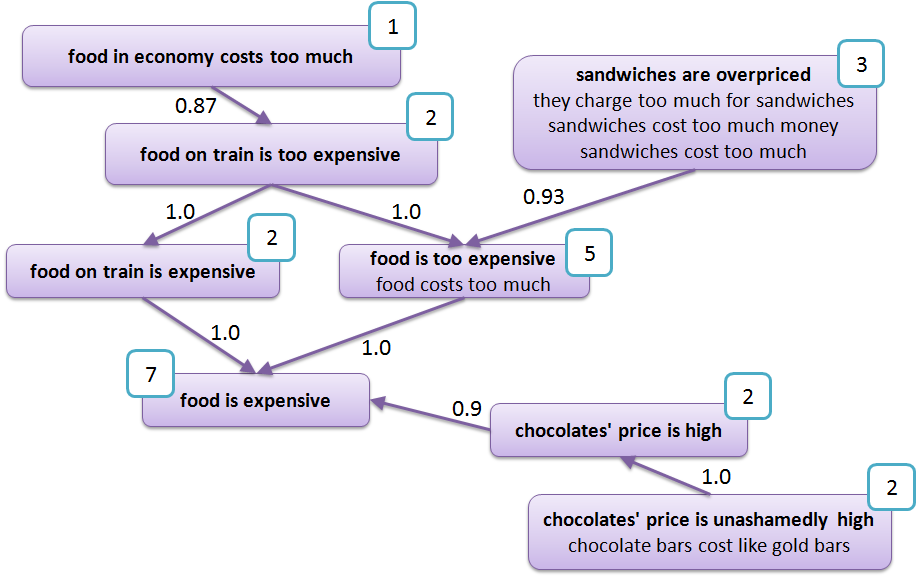
The edges of the collapsed graph are collapsed entailment relations (*EntailmentRelationCollapsed*).

#### class EntailmentGraphCollapsed (eu.excitementproject.tl.structures.collapsedgraph)

The structure of the collapsed graph is simpler than that of the raw graph:

* + There is no need in multiple edges between the same pair of nodes: such multiple edges are collapsed to form a single edge with the final entailment decision. The presence of an edge between two nodes (source and target) means that there is an entailment relation between the two nodes in the direction source -> target.
  + There are no cycles: entailment paths, which form cycles, are collapsed to form equivalence class (paraphrase) nodes.
  + Edges and nodes contain less information: the information needed for internal purposes of building the graph is excluded.

The example below presents a simple collapsed graph. Numbers attached to the nodes are counters of occurrences.



This graph is built from the raw graph, by collapsing multiple edges between the same pair of vertices into one edge, and grouping entailment units into equivalence classes. This process is performed by the [Collapsed Graph Generator](#_Collapsed_graph_generator) module.

Unlike the raw graph, this is no longer a multigraph, but a simple directed graph. It extends DefaultDirectedWeightedGraph, for inherited methods see the JavaDoc: <http://jgrapht.org/javadoc/org/jgrapht/graph/DefaultDirectedWeightedGraph.html>

* Attributes:
  + *Set<java.lang.String> textualInputs* – the textual inputs (complete statements), on which the entailment graph was built..
  + *int numberOfEntailmentUnits* – the number of entailment units contained in the graph. This number is not necessarily the same as the number of nodes in the graph, since each equivalence class node corresponds to one or more entailment unit(s).
* Constructors:

*EntailmentGraphCollapsed()*

The constructor generates an empty raw entailment graph.

*EntailmentGraphCollapsed(java.io.File xmlFile)*

The constructor uploads an existing collapsed graph from the given xml file. The file is assumed to follow the format of the output produced by toXML() method of the EntailmentGraphCollapsed class.

* + - @param xmlFile – the xml file from which to load a previously produced raw entailment graph.
    - throws EntailmentGraphCollapsedException if creating the graph from the given file did not succeed for some reason (file not found, wrong xml format etc.)
* Methods:

*EquivalenceClass getVertex(java.lang.String text)*

*EquivalenceClass getVertex(EntailmentUnit eu)*

* + - @return (EquivalenceClass) – the node, which corresponds to the given text or to the given entailment unit. If such a node is not found in the graph, returns null.
    - @param text – the canonical text of one of the entailment units included in the node to be returned
    - @param eu – the entailment unit included in the node to be returned

*int getNumberOfInteractions()*

* + - @return (*int*) – returns the number of interactions, based on which the entailment graph was built.

*List<EquivalenceClass> sortNodesByNumberOfInteractions(int X)*

This method is required by the NICE scenario and returns the top X nodes in the graph, based on the number of associated interactions.

* + - @return (List<EquivalenceClass>) – the list of equivalence class nodes, sorted in descending order by the number of associated interactions.

*int getNumberOfEntailmentUnits()*

This method is required by the NICE scenario.

* + - @return (int) – returns the number of entailment units contained in the graph.

*int getNumberOfEquivalenceClasses()*

This method is required by the NICE scenario.

* + - @return (int) – returns the number of nodes in the graph.

*int getNumberOfFragmentGraphs()*

This method is required by the NICE scenario.

* + - @return (int) – returns the number of fragment graphs based on which the collapsed graph was built.

*Set<EntailmentUnit> getEquivalentEntailmentUnits(java.lang.String entailmentUnitText)*

*Set<EntailmentUnit> getEquivalentEntailmentUnits(EntailmentUnit entailmentUnit)*

This method returns equivalent entailment units for a given input entailment unit, i.e. entailment units, which are in the same equivalence class.

* + - @return (Set<EntailmentUnit>) – the set of entailment units. If there are no entailment units answering this search, an empty set is returned. If there is no such entailment unit as given in the method’s parameter, the method returns null.
    - @param entailmentUnitText or entailmentUnit – the canonical text of the entailment unit whose paraphrases are to be found or the entailment unit itself.

*Set<EquivalenceClass> getEntailingEquivalenceClasses(java.lang.String entailmentUnitText)*

*Set<EquivalenceClass> getEntailingEquivalenceClasses(EntailmentUnit entailmentUnit)*

This method returns equivalence classes containing entailment units entailing the input entailment unit, i.e. equivalence classes, for which there is an edge going from this equivalence class to the equivalence class of the input entailment unit.

* + - return (Set<EquivalenceClass>) – the set of equivalence class nodes. If there are no equivalence classes answering this search, an empty set is returned. If there is no such entailment unit as given in the method’s parameter, the method returns null.
    - @param entailmentUnitText or entailmentUnit – the canonical text of the entailment unit whose entailing equivalence classes are to be found, or the entailment unit itself.

*Set<EquivalenceClass> getEntailedEquivalenceClasses(java.lang.String entailmentUnitText)*

*Set<EquivalenceClass> getEntailedEquivalenceClasses(EntailmentUnit entailmentUnit)*

This method returns equivalence classes containing entailment units entailed by the input entailment unit, i.e. equivalence classes for which there is an edge going to this equivalence class from the equivalence class of the input entailment unit.

* + - return (Set<EquivalenceClass>) – the list of equivalence class nodes. If there are no equivalence classes answering this search, empty set is returned. If there is no such entailment unit as given in the method’s parameter, the method returns null.
    - @param entailmentUnitText or entailmentUnit – canonical text of the entailment unit whose entailed equivalence classes are to be found, or the entailment unit itself.

*EntailmentGraphCollapsed getSubgraphFor(java.lang.String entailmentUnitText)*

*EntailmentGraphCollapsed getSubgraphFor(EntailmentUnit entailmentUnit)*

This method is required by the ALMA scenario and returns a subgraph with all nodes containing the input entailment unit, as well as all nodes directly connected to one of these nodes, i.e., all equivalent, entailed or entailing entailment units.

* + - @return (EntailmentGraphCollapsed) – the required subgraph. If there are no entailment units answering this search, empty graph is returned (graph with no nodes and no edges). If there is no such entailment unit as given in the method’s parameter, the method returns null.
    - @param entailmentUnitText or entailmentUnit – canonical text of the entailment unit whose subgraph should be returned, or the entailment unit itself.

*Set<String> getRelevantInteractionIDs(java.lang.String entailmentUnitText)*

*Set<String> getRelevantInteractionIDs(EntailmentUnit entailmentUnit)*

This method is required by the ALMA scenario and returns the ids of

interactions that contain an entailment unit that is equivalent to the input entailment unit based on the entailment graph.

* + - @return (Set<String>) – the set of relevant interactions’ ids. If there is no such entailment unit as given in the method’s parameter, the method returns null.
    - @param entailmentUnitText or entailmentUnit – canonical text of the entailment unit whose relevant interactions need to be found, or the entailment unit itself.

Methods to output the graph:

*void toXML(java.lang.String filename)*

The method saves the graph to an xml file with the given filename.

* + - @param filename – the name of the xml file to which the graph should be saved.
    - throws EntailmentGraphCollapsedException if saving the graph to the given file did not succeed for some reason.

*java.lang.String toDOT()*

*void toDOT(java.lang.String filename)*

The method generates a single string, which contains the graph in DOT format (http://en.wikipedia.org/wiki/DOT\_(graph\_description\_language)) for visualization. The string is either returned or saved to the given output file (if provided).

* + - @return (*java.lang.String*) – the string with the graph in DOT format.
    - @param filename – the name of the output file. If provided, the string will be saved to the file rather than returned.
    - throws IOException if saving to the file failed for some reason.

Specialized implementation of graph node methods

*toString()*

#### class EquivalenceClass (eu.excitementproject.tl.structures.collapsedgraph)

The node of the collapsed entailment graph is an equivalence class. This type of node contains all text fragments that are equivalent from the point of view of textual entailment.

* Attributes:
  + *Set<EntailmentUnit> entailmentUnits* – the set of all entailment units, which were considered paraphrasing
  + *String label* – the representative text
  + *int mentionsNumber*  – overall number of entailment unit mentions for the equivalence class (sum of the frequencies of all the entailment units, which form the equivalence class)
* Constructors:

*EquivalenceClass(EntailmentUnit eu)*

Constructs the equivalence class based on a single entailment unit

* + - @param eu – the entailment unit

*EquivalenceClass(Set<EntailmentUnit> s\_eu)*

Constructs the equivalence class based on a set of entailment units

* + - @param s\_eu – the set of entailment units

*EquivalenceClass(java.lang.String label, Set<EntailmentUnit> s\_eu)*

Constructs the equivalence class based on a set of entailment units with an explicitly specified label (representative text)

* + - @param label – the label of the equivalence class
    - @param s\_eu – the set of entailment units
* Methods:

*Set<EntailmentUnit> getEntailmentUnits()*

Returns the set of entailment units, which form the equivalence class

* + - @return (Set<EntailmentUnit>) – the set of entailment units

*Set<String> getEntailmentUnitTexts()*

Returns the set of the texts of entailment units, which form the equivalence class

* + - @return (Set<java.lang.String>) – the set of entailment unit texts

*boolean containsEntailmentUnit(EntailmentUnit eu)*

The method returns true if the input entailment unit is contained in the equivalence class. Otherwise it returns false.

* + - @param eu - the input entailment unit
    - @return true/false

*String getLabel()*

Returns the label of the equivalence class

* + - @return (java.lang.String) – the label

*int getNumberOfMentions()*

Returns the number of mentions of the equivalence class (sum of the number of mentions of each of its entailment units)

* + - @return (int) – the number of mentions

*Set<String> getInteractionIDs()*

Returns the ids of interactions that contain entailment units, which are part of the equivalence class

* + - @return (Set<String>) – the set of interaction ids

*void add (EntailmentUnit eu)*

*void add (Set<EntailmentUnit> s\_ eu)*

Adds the input entailment unit(s) to the equivalence class. If the set already contains any of the input entailment units, the corresponding entailment unit will not be added. The method does not change the representative text (label) of the equivalence class.

* + - @param eu – the entailment unit to be added
    - @param s\_eu – the set of entailment units to be added.

Specialized implementation of graph node methods: *toString()*

#### class EntailmentRelationCollapsed (eu.excitementproject.tl.structures.collapsedgraph)

This class implements the collapsed graph edges, obtained by collapsing multiple edges (decisions from different EDAs or other sources) from the raw graph into one edge. The edges are directional and represent entailment relations between the source and the target nodes (source -> target). This class extends DefaultEdge: <http://jgrapht.org/javadoc/org/jgrapht/graph/DefaultEdge.html>

* Attributes:
  + *EquivalenceClass* *source* – the source node
  + *EquivalenceClass* *target* – the target node
  + *double confidence* – confidence that entailment relation source -> target indeed holds
* Constructors:

*EntailmentRelationCollapsed(EquivalenceClass source, EquivalenceClass target, double confidence)*

Creates an edge with the given confidence from source to target.

* + - @param source – the source node
    - @param target – the target node
    - @param confidence – the confidence
* Methods:

*double getConfidence()*

Returns the confidence of the edge.

* + - @return (double)

Specialized implementation of graph edge methods:

*getSource(), getTarget(), toString()*

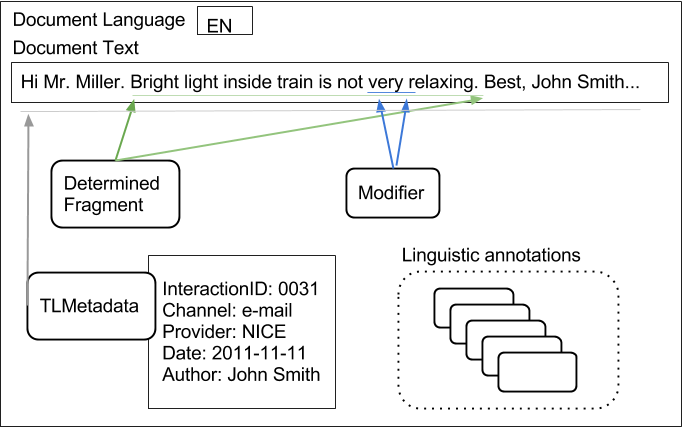
# UIMA Type System for Transduction Layer

## Introduction

There are two types of CASes used in the Transduction Layer. The first is the *input CAS*, which holds the customer interaction, fragment annotations and modifier annotations. The steps for obtaining these annotations are detailed in the decomposition data flow description.

The other type is the CAS data that is used as input for the EDA.

In this section, we present only the type systems that are newly added for the first case (input CAS). For other types (linguistic annotations for EDAs), please refer to the Excitement open platform specification. One input CAS always holds one Interaction. It can have multiple numbers of fragments and modifiers. The CAS can only have one Metadata, which includes the interaction ID. The CAS will also contain all linguistic annotations (POS, lemma, parse result, etc). The following figure shows an example of an input CAS.



Details about all types related to the input CAS are given in the following section.

## Types

### Metadata (eu.excitement.type.tl)

#### Supertype

- uima.tcas.Annotation

#### Features

The type includes: (all strings)

* interactionId
* channel
* provider
* date (string as YYYY-MM-DD)
* businessScenario
* author
* category

#### Description

This type description file defines the input CAS Metadata type, which records various metadata related to the Interaction and the input CAS. Note that one CAS should have only one metadata (only the first one should be considered, if more than one), and each CAS should have one metadata, even if all of its fields are null. Note that language ID is not recorded in this metadata type. It is directly recorded in CAS. Also note that all of the metadata are simply strings, and can be null if that metadata is missing.

### FragmentAnnotation (eu.excitement.type.tl)

#### Super type

-uima.tcas.Annotation

#### Features

- begin/end (inherited): this span covers the general region, even if the fragment text is non-contiguous within the region.

- text (String): this holds the text that this fragmentation represents.

- fragParts (Array of FragmentParts type): this holds one or more FragmentsParts type in an array. Thus, it can actually map non-contiguous regions. If the fragmentation is contiguous, this array has only one item.

#### Description

This type annotates a *fragment*, as defined in EXCITEMENT WP6 (and WP2). This is the base type of two different fragments: AssumedFragment type and DeterminedFragment type.

Example.

0 23 44 67

The connection was slow. I was on vacation. GPRS was specially slow.

begin:0

end:67

text: The connection was slow. GPRS was specially slow.

fragParts(0): FragmentParts -begin:0 -end:23

(1): FragmentParts -begin:44 -end:67

### FragmentPart (eu.excitement.type.tl)

#### Super type

-uima.tcas.Annotation

#### Features

(none)

#### Description

This is a type that is designed to represent one contiguous region of a (potentially non-contiguous) fragment. This is only used for that purpose, and does not have any additional feature.

### AssumedFragment (eu.excitement.type.tl)

#### Super type

- FragmentAnnotation

#### Features

(none)

#### Description

This is a fragment annotation that is used to mark the *assumed fragment*. The WP7 application layer uses this annotation to mark what the WP7 application considers a possible fragment. This might not be accurate, and WP6 performs an additional fragment analysis.

### DeterminedFragment (eu.excitement.type.tl)

#### Super type

- FragmentAnnotation

#### Features

(none)

#### Description

This is a fragmentation annotation that is "determined" by WP6 internal modules. Unlike the "assumed fragment", this is the actual fragment that will be treated as the real fragment.

### ModifierAnnotation (eu.excitement.type.tl)

#### Super type

-uima.tcas.Annotation

#### Features

- modifierParts (Array of ModifierPart type): this holds one or more ModifierPart type in an array. Thus, it can actually map non-contiguous regions. If the modifier is contiguous, this array only has one item.

- dependsOn (ModifierAnnotation): If this modifier depends on some other modifier, this feature points to that modifier. This modifier depends on the pointed to modifier. (If the modifier pointed to by this feature does not exist, this modifier is not grammatical / meaningless.)

#### Description

This annotation type annotates a region as a "modifier", and it is used within WP6 modules to create the fragment graph nodes: each node represents a unique (and valid, from the point of view of the dependsOn relation) combination of modifiers. While this could be simple, it gets a bit complicated by "dependsOn" and "non-contiguous" regions.

See the following example:

0 24 27

Seats are uncomfortable as too old.

Here we have two modifiers.

ModifierAnnotation #1 "too"

-begin: 27

-end: 29

-modifierParts: (0) -begin:27 -end:29

-dependsOn: ModifierAnnotation #2

ModifierAnnotation #2 "as ... old"

-begin: 24

-end: 34

-modifierParts: (0) -begin:24 -end:25

(1) -begin:31 -end:33

-dependsOn: (null)

The above example shows two modifiers that have a dependency relation: “too” has meaning only if “old” is present. Thus, removing only modifier #2, is not possible. This is marked in #1 that it depends on #2. (#1 is not a valid stand-alone modifier, and if #2 is removed, #1 should also be removed).

### CategoryAnnotation (eu.excitement.type.tl)

#### Super type

- FragmentAnnotation

#### Features

- categories (array of CategoryDecision type): at least one or more category decision data associated with this fragment.

#### Description

This is a type designed to represent the result of use case 2 processing. It represents a fragment, but also with the data associated for category decision. The fragment annotated by this type has one or more category decision types, which annotates category id and confidence for that category.

### CategoryDecision (eu.excitement.type.tl)

#### Super type

-uima.cas.Top

#### Features

- categoriy id (String)

- confidence (Float)

#### Description

This is the metadata used for the output of use case 2 (category annotation). This type is used in CategoryAnnotation, as an element of an array.

# Interface Definitions for the WP6 Modules

This section describes interfaces of all modules. Before looking into the interface definitions, please take a look at the section 2, which details how each component listed in the section contributes to the whole process.

## Interfaces of Decomposition Components

### Fragment Annotator Module: interface *FragmentAnnotator* (eu.excitementproject.tl.decomposition.api)

#### General Description

As described in section 2.1.2 in the data flow chapter, an instance of *FragmentAnnotator* annotates a part of an interaction text as a coherent statement (a *fragment*). One fragment holds one statement. The goal of a fragment annotator is to identify them and correctly mark them.

The TL type system of input CAS has two types of fragment annotations. One is [*assumedFragment*](#_AssumedFragment_(eu.excitement.type) and the other is [*determinedFragment*](#_DeterminedFragment_(eu.excitement.t). An assumed fragment represents the belief of the caller who prepared the input, while *determinedFragment* represents the decision of the *FragmentAnnotator* (an instance of this interface). The fragment annotator may use and rely on the assumed fragment annotation as evidence, or feature. But in general, it does not blindly follow it. *determinedFragment* annotations are the output of this module, and will be used as the “real” fragments in the downstream modules.

Thus, Fragment Annotator consumes as its input an interaction and optional assumed fragment annotations. The output is determined fragments, which work as the final decision of fragments.

#### API Methods

The interface contains one method, for adding (determined) fragment annotation to the given CAS.

* *void annotateFragments(JCas text) throws FragmentAnnotatorException*
  + @param text – JCas with text and metadata. It may additionally hold assumedFragment annotation.
  + @return – the method returns nothing. But the argument JCas text is enriched with real (determined) fragment annotations.
  + @throws (FragmentAnnotatorException) if any needed data is missing in the JCas, or if the module cannot successfully annotate the determined fragment.

#### Related Data Structure & Other Notes

Each fragment annotation can be non-contiguous (check the type definition in section 4.2.2 to see how this is represented in the annotation.)

The specific implementations of this module may call an LAP pipeline. The implementations of this interface can be found under the *eu.excitementproject.tl.decomposition.fragmentannotator* package. For information on the implementation(s) of this module see section 6.1.1.

Any new implementation of this module should consider extending the *AbstractFragmentAnnotator* class. It forces the implementation to expose LAP in the constructor. Also, any additional configurable parameters of this module implementation should be clearly exposed in the constructor.

### Modifier Annotator Module: interface *ModifierAnnotator* (eu.excitementproject.tl.decomposition.api)

#### General Description

As described in section 2.1.3, a modifier annotator annotates any “non-essential” part of a statement (fragment) as *modifier*. The term may be a bit misleading, since modifier does not necessarily refer to a syntactic modifier, but rather semantically to a term, which modifies the meaning of the predicate of the statement or the meaning of its main arguments (non-essential arguments, conditions, etc.). By using this term we follow the terminology defined in WP2. Examples of modifiers can be found in section 3.3.1, as well as in WP2 documentation.

A modifier annotator gets one input CAS, which holds the corresponding interaction with fragment annotation, and annotates all modifiers found in the interaction. It uses the TL type ModifierAnnotation to do this.

#### API Methods

The interface contains one method, for adding modifier annotation to the given CAS.

* *void annotateModifiers(JCas text) throws ModifierAnnotatorException;* 
  + @param text – a JCas with interaction text, metadata and determined fragment annotation(s). When successfully run, the input CAS is enriched with modifier annotation(s).
  + @return – the method returns nothing. The input CAS is directly enriched.
  + @throws ModifierAnnotatorException if any of the needed data is missing in the input CAS, or when the module could not annotate the modifiers due to some failures.

#### Related Data Structure & Other Notes

Each modifier annotation can be non-continuous, and may contain dependencies among modifiers (check the type definition in section 4.2.6 for some examples.)

Specific implementations of this module may need to call an LAP pipeline. The implementations of this interface can be found under the *eu.excitementproject.tl.decomposition.modifierannotator* package. For information on the implementation(s) of this module see section 6.1.2.

New implementations of this module should consider extending *AbstractModifierAnnotator* class. It forces the implementation to expose LAP in the constructor. Also, any additional configurable parameters of this module implementation should be clearly exposed in the constructor.

### Fragment Graph Generator Module: interface *FragmentGraphGenerator* (eu.excitementproject.tl.decomposition.api)

#### General Description

*FragmentGraphGenerator* is the interface between user-provided document CAS objects and the Transduction Layer. The interface produces a set of *FragmentGraph*-s from an input CAS object. For each fragment (*determinedFragment*) annotated in the CAS, there will be a *FragmentGraph* object, which is further processed within the platform.

#### API Methods

The interface contains one method, for generating the FragmentGraph structures from the user’s input:

* *Set<FragmentGraph> generateFragmentGraphs(JCas text)*:
  + @param text – the CAS object representing a document/user interaction
  + @return (Set<FragmentGraph>) – the set of FragmentGraph objects, one for each of the DeterminedFragment annotations in the text
  + @throws FragmentGraphGeneratorException when the FragmentGraph generation failed

#### Related Data Structure & Other Notes

The implementations of this interface can be found under the *eu.excitementproject.tl.decomposition.fragmentgraphgenerator* package.

New implementations of this module should consider extending the *AbstractFragmentGraphGenerator* class. For information on the implementation(s) of this module see section 6.1.3.

## Interfaces of Composition Components

### Graph Merger Module: interface *GraphMerger* (eu.excitementproject.tl.composition.api)

#### General Description

The main goal of this module is “enriching” a raw graph (*EntailmentGraphRaw*), by merging newly mined fragment graphs. Conceptually, its input is two things. One is a set of fragment graphs, and the other is the raw graph. After a successful run, the module returns the enriched entailment graph. This module uses the entailment decision capability (EDA) of the EXCITEMENT open platform (EOP) to add fragment graph into the entailment graph. For more details see section 2.2.1.

#### API Methods

*EntailmentGraphRaw mergeGraphs( Set<FragmentGraph> fragmentGraphs, EntailmentGraphRaw workGraph)*

The method consumes a raw graph (*EntailmentGraphRaw*) and a set of fragment graphs, which should be merged with the given raw graph. In case of success the enriched raw graph is returned. Otherwise the method throws a *GraphMergerException*.

* + @param fragmentGraphs – a set of fragment graphs. If the set is empty or null, the input raw graph is returned unchanged.
  + @param workGraph – the raw entailment graph that should be enriched. If this parameter is null, a new empty graph is created and merged with the given fragmentGraphs.
  + @return (EntailmentGraphRaw) the given raw graph enriched by the given set of fragments.
  + @throws (GraphMergerException) if the implementation cannot merge the graphs for some reason

*EntailmentGraphRaw mergeGraphs(FragmentGraph fragmentGraph, EntailmentGraphRaw workGraph)*

The method consumes a raw graph (*EntailmentGraphRaw*) and a single fragment graph, which should be merged with the given raw graph. In case of success the enriched raw graph is returned. Otherwise the method throws a *GraphMergerException*.

* + @param fragmentGraph – the fragment graph. If this parameter is null, the input raw graph is returned unchanged.
  + @param workGraph – the raw entailment graph that should be enriched. If this parameter is null, a new raw graph is created based on the given fragmentGraph.
  + @return (EntailmentGraphRaw) the given raw graph enriched by the given fragment graph.
  + @throws (GraphMergerException) if the implementation cannot merge the graphs for some reason

#### Related Data Structure & Other Notes

An implementation of this interface might need to call LAP and is most likely to call EDA. The needed LAP and EDA related configurations should be passed via the Constructor (thus, they are not defined in the interface). Also, any additional configurable parameters of this module implementation should be clearly exposed in the constructor.

The implementations of this interface can be found under the *eu.excitementproject.tl.composition.graphmerger* package.

For implementing this interface it is recommended to extend the *AbstractGraphMerger* class. This abstract implementation contains auxiliary methods that are expected to be common over different implementations. For information on the implementation(s) of this module see section 6.2.1.

### Collapsed Graph Generator Module: interface *CollapsedGraphGenerator* (eu.excitementproject.tl.composition.api)

#### General Description

This module consumes a raw graph (*EntailmentGraphRaw*) and produces the *collapsed graph* or final graph (*EntailmentGraphCollapsed*). For more details see section 2.2.2.

#### API Methods

*EntailmentGraphCollapsed generateCollapsedGraph(EntailmentGraphRaw workGraph)*

*EntailmentGraphCollapsed generateCollapsedGraph(EntailmentGraphRaw rawGraph, double threshold)*

The method consumes a raw graph (*EntailmentGraphRaw*) and produces the final collapsed entailment graph (*EntailmentGraphCollapsed*). In case of success the collapsed graph is returned. Otherwise the method throws a *CollapsedGraphGeneratorException*.

* + @param workGraph – the raw entailment graph, which should be collapsed
  + @param threshold –if provided, confidence threshold representing the minimum confidence for an edge from the raw graph to be kept in the collapsed graph
  + @return (EntailmentGraphCollapsed) the resulting collapsed entailment graph
  + @throws CollapsedGraphGeneratorException if the implementation cannot convert the graph for some reason

#### Related Data Structure & Other Notes

We do not foresee any external EOP component dependency for this module. Yet, if any arguments or configurable values are needed, they should be exposed in the implementation constructor.

The implementations of this interface can be found under the *eu.excitementproject.tl.composition.collapsedgraphgenerator* package.

For implementing this interface it is recommended to extend the *AbstractCollapsedGraphGenerator* class. This abstract implementation contains auxiliary methods that are expected to be common over different implementations.

For information on the implementation(s) of this module see section 6.2.2.

### Node Matcher Module: interface *NodeMatcher* (eu.excitementproject.tl.composition.api)

#### General Description

This module matches a given *FragmentGraph* against a given *EntailmentGraphRaw* and returns a set of *NodeMatch-*es. For more details see section 2.3.1.

#### API Methods

The interface contains one method:

*Set<NodeMatch> findMatchingNodesInGraph(FragmentGraph fragmentGraph, EntailmentGraphRaw entailmentGraph) throws NodeMatcherException*

This method compares a fragment graph to a raw graph and returns a set of node matches.

* + @param fragmentGraph – the fragment graph to be matched
  + @param entailmentGraph – the raw entailment graph to be matched against
  + @return (Set<NodeMatch>) – the set of node matches
  + @throws NodeMatcherException if the match fails

#### Related Data Structure & Other Notes

There are two related data structures: NodeMatch and PerNodeScore, which are defined in the following.

The implementations of this interface can be found under the *eu.excitementproject.tl.composition.nodematcher* package.

For implementing this interface it is recommended to extend the *AbstractNodeMatcher* class. This abstract implementation contains auxiliary methods that are expected to be common over different implementations. For information on the implementation(s) of this module see section 6.2.3.

##### Class NodeMatch

A node match holds an *EntailmentUnitMention* associated to a set of *PerNodeScore*-s.

* Attributes:
* *EntailmentUnitMention mention* – the mention within the fragment graph, for which one or more matches was found
* *Set<PerNodeScore> scores* – a set of per node scores
* Methods (getters and setters only):

*EntailmentUnitMention getMention()*

* + @return mention

*void setMention(EntailmentUnitMention mention)*

* + @param mention

*Set<PerNodeScore> getScores()*

* + @return scores

*void setScores(Set<PerNodeScore> scores)*

* + @param scores

##### Class PerNodeScore

A “per node score” keeps matched nodes with the corresponding confidence score of the match. It is a tuple <E,C>, where E denotes an *EntailmentUnit* (a matching node in the raw graph) and C denotes a confidence score (a score expressing how well this node matches the entailment unit mention, to which the *PerNodeScore* object is associated).

* Attributes:
* *EntailmentUnit node* – a matching raw graph node
* *double score* – confidence of the match
* Methods (getters and setters only):

*EntailmentUnit getNode()*

* + @return node

*void setNode(EntailmentUnit node)*

* + @param node;

*double getScore()*

* + @return score

*void setScore(double score)*

* + @param score

### Category Annotator Module: interface *CategoryAnnotator* (eu.excitementproject.tl.composition.api)

#### General Description

This module adds category annotation to a given input CAS, i.e. it assigns, to a particular fragment in the input CAS, a category ID together with a confidence score expressing how well this category matches the fragment. For computing this confidence score, this module makes use of the category information stored in the *NodeMatch* objects returned for the fragment, i.e. the output of the NodeMatcher module. *NodeMatch* objects hold category information within their *PerNodeScore* objects: Each *PerNodeScore* holds a matching raw graph node with entailment unit mentions. Each mention is associated to the category of the interaction it was extracted from. The goal of the module is to combine the category information of the different mentions associated to the raw graph nodes matching the input fragment into a single confidence score for each category and fragment. For more details see section 2.3.2.

#### API Methods

The interface contains one method:

*void addCategoryAnnotation(JCas cas, Set<NodeMatch> matches) throws CategoryAnnotatorException*

This method takes a set of node matches and combines the category information found in the node matches to a category confidence that is then added to the input CAS.

* + @param cas – input CAS
  + @param matches – set of node matches
  + @return no new data, but the input CAS is annotated with category annotation
  + @throws (CategoryAnnotatorException) if category annotation fails

#### Related Data Structure & Other Notes

The implementations of this interface can be found under the *eu.excitementproject.tl.composition.categoryannotator* package.

For implementing this interface it is recommended to extend the *AbstractCategoryAnnotator* class. This abstract implementation contains methods that are expected to be common over different implementations. For information on the implementation(s) of this module see section 6.2.4.

## Top Level Interface Definition

### Introduction to the Top Level

In this document, the top level is used for the main data flow runner. Transduction Layer (TL) top level code configures and runs available components of the TL to instantiate one instance of the TL that will work for a WP7 use case.

The TL top levels are designed to make the TL transparent: Once a WP7 user sets the TL and EOP up and running, WP7 code only needs to access the top level APIs to get all the results.

Note that, to the TL layer and to the users (WP7), EOP is exposed with two interfaces. One is the LAPAccess interface that exposes annotation capabilities, and the other is the EDABasic interface that enables us to make entailment decisions.

EOP accepts and uses a configuration system. EDAs are often shipped with a sophisticated and already configured configuration file. Such files must be provided at the initialization time of EDAs (also for LAPs, if LAP is complex). To customize EDAs of the EOP to a specific need (e.g., retraining, parameter tuning), one has to study the configuration file of a specific EDA.

Note that within the TL layer, we have adopted a "configuration-less" approach of providing modules. The TL layer does not keep its own configuration files. However, it is definitely not a stateless machine: it has various possible parameters (e.g., threshold variables). However, those are (or will be) systemically exposed on each component's constructor. And then, they will also be exposed in the top-level constructors’ arguments.

The reason for this "configuration-less" approach, in which we expose every state/parameter on the constructor level, is to reduce the amount of configuration storage. We expect that the calling system of WP7 will have its own property/configuration/state storage

mechanism, differing for each industrial partner. By exposing and not defining parameters as a configuration format, we hope to remove the need of WP7 users to work with three configurations (industrial system, EOP, and then TL). We aim to be the adaptor layer,

in which the storage of properties/configuration can be integrated into the industrial partners’ system by exposing all TL properties transparently to the users.

The TL currently provides two top level APIs described in the following sections. One is for use case 1, and the other is for use case 2. Any top level module should implement one or both of the interfaces.

### Use Case 1 Top Level API: interface *UseCaseOneRunner* (eu.excitementproject.tl.toplevel.api)

#### General Description

This top level interface configures and runs the TL platform for WP7’s use case 1, applicable to all the scenarios.

#### API Methods

This use case requires building raw and collapsed entailment graphs from user interactions, and is implemented through the following methods:

* *EntailmentGraphRaw buildRawGraph (List<JCas> annotatedInteractions)*
  + @param interactions – a set of (annotated) user interactions represented as CAS objects
  + @return (EntailmentGraphRaw) – raw entailment graph of text fragments connected through entailment relations; the graph is obtained by merging the *FragmentGraph*-s corresponding to each fragment (*DeterminedFragment*) annotation in the input CAS objects.
* *EntailmentGraphRaw buildRawGraph (Set<Interaction> interactions)*
  + @param interactions – a set of user interactions represented as Interaction objects
  + @return (EntailmentGraphRaw) – raw entailment graph of text fragments connected through entailment relations; the graph is obtained by merging the *FragmentGraph*-s corresponding to each fragment (*DeterminedFragment*) annotation in the input CAS objects into a raw graph, and further collapsing the raw graph based on the (optional) confidence score.
* *EntailmentGraphCollapsed buildCollapsedGraph (List<JCas> annotatedInteractions)*
* *EntailmentGraphCollapsed buildCollapsedGraph (List<JCas> annotatedInteractions, double threshold)*
  + @param interactions – a set of (annotated) user interactions represented as CAS objects
  + @param threshold – if provided, confidence threshold representing the minimum confidence for an edge from the raw graph to be kept in the collapsed graph
  + @return (EntailmentGraphCollapsed) – graph of text fragments connected through entailment relations, obtained by by merging the *FragmentGraph*-s corresponding to each fragment (*DeterminedFragment*) annotation in the *Interaction* objects into a raw graph, and further collapsing the raw graph based on the (optional) confidence score.
* *EntailmentGraphCollapsed buildCollapsedGraph (Set<Interaction> interactions)*
* *EntailmentGraphCollapsed buildCollapsedGraph (Set<Interaction> interactions, double threshold)*
  + @param interactions – a set of user interactions represented as Interaction objects
  + @param threshold – if provided, confidence threshold representing the minimum confidence for an edge from the raw graph to be kept in the collapsed graph
  + @return (EntailmentGraphCollapsed) – graph of text fragments connected through entailment relations, obtained by collapsing an *EntailmentGraphRaw* based on the confidence score; the raw graph was obtained by annotating the input interactions and merging the *FragmentGraph*-s corresponding to each fragment (*DeterminedFragment*) annotation in the *Interaction* objects.
* *EntailmentGraphCollapsed buildCollapsedGraph (File rawGraph)*
* *EntailmentGraphCollapsed buildCollapsedGraph (File rawGraph, double threshold)*
  + @param rawGraph – a (XML) file representation of a raw graph
  + @param threshold – if provided, confidence threshold representing the minimum confidence for an edge from the raw graph to be kept in the collapsed graph
  + @return (EntailmentGraphCollapsed) – graph obtained by collapsing the *rawGraph* based on the confidence score

#### Related Data Structure & Other Notes

The implementations of this interface can be found under the *eu.excitementproject.tl.toplevel.usecaseonerunner* package.

### Use Case 2 Top Level API: interface *UseCaseTwoRunner* (eu.excitementproject.tl.toplevel.api)

#### General Description

This top level interface configures and runs available TL components to instantiate one “instance” of the transduction layer that will work for WP7’s use case 2.

#### API Methods

* *void annotateCategories(JCas cas, EntailmentGraphRaw graph)*
  + @param cas – input CAS
  + @param graph – raw entailment graph created for this domain
  + @return no new data, but the input CAS is annotated with category annotation

#### Related Data Structure & Other Notes

The implementations of this interface can be found under the *eu.excitementproject.tl.toplevel.usecasetworunner* package.

# Implementation of the Prototype

In the previous chapters, we have provided a detailed description of the data structures and interfaces we defined for the various transduction layer modules, including the core modules as well as the top level modules. In this chapter, we describe prototypical implementations of those modules, which show how the defined interfaces and data structures can be used to realize the two industrial use cases. We also show how they can be used to process all three languages and all three data channels that are part of the project.

## Implementation of Decomposition Components

### Fragment Annotator Module: class SentenceAsFragmentAnnotator (eu.excitementproject.tl.decomposition.fragmentannotator)

This class implements the interface *FragmentAnnotator*. This is a very simple fragment annotation method that annotates each sentence as a separate fragment. The component calls the given LAP or searches the input CAS to find the sentence annotation, and then annotates each sentence as one fragment.

The component calls the given LAP once, if no sentence annotation was found in the input CAS. If one or more sentences were found within the CAS, LAP call is not performed. The component raises an exception if the given LAP cannot produce sentence annotation (i.e., if the module cannot find any sentence).

### Modifier Annotator Module: class AdvAsModifierAnnotator (eu.excitementproject.tl.decomposition.modifierannotator)

This class implements the interface *ModifierAnnotator*. This simple modifier annotation method relies solely on Part-Of-Speech (POS) information. This module annotates all adverbs as modifiers.

The component raises an exception if the given LAP cannot produce POS annotations (thus the module has no POS information).

The component calls the given LAP once, if no POS annotation can be found in the given CAS. If it finds one or more POS annotations within the CAS, no LAP call is performed.

### Fragment Graph Generator Module: class FragmentGraphGeneratorFromCAS (eu.excitementproject.tl.decomposition.fragmentgraphgenerator)

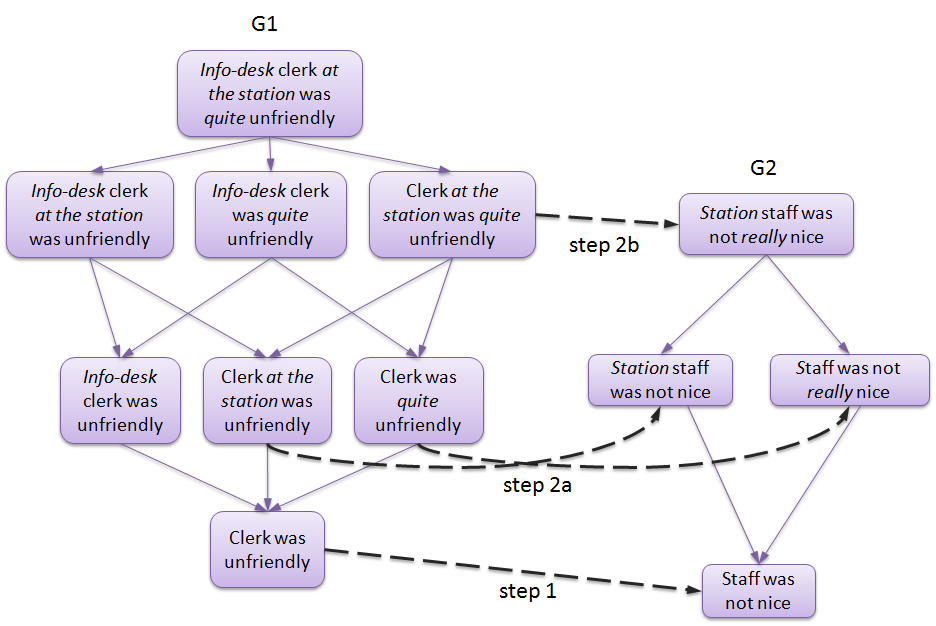
The *FragmentGraphGeneratorFromCAS* class implements the *FragmentGraphGenerator* interface, where we assume the input data is a CAS object with the structure described in the chapter 4 of this document. It implements the *generateFragmentGraphs* method of the interface (described in section 5.1.3), by iterating over the *DeterminedFragment* annotations from the input CAS object.

## Implementation of Composition Components

### Graph Merger Module: class AutomateWP2ProcedureGraphMerger (eu.excitementproject.tl.composition.graphmerger)

This implementation of the *GraphMerger* module automates the manual annotation procedure developed within WP2. According to this procedure, in order to merge two fragment graphs G1 and G2, the following steps should be performed:

1. Check for entailment (obtain EDA decision) in either direction between the base statements of the two graphs. If there is no entailment relation, the graphs should not be connected.
2. If there is entailment in one direction (let’s assume G1 base statement -> G2 base statement)
   1. Check for entailment between all the pairs G1 node1 -> G2 node1, where nodes1 hold 1-modifier statements, which directly entail the base statement of the corresponding graph.
   2. Induce entailment for upper-level (i=2..n) nodes in the direction G1 nodei -> G2 nodei, if each of the nodes G1 nodei-1 directly entailed by G1 nodei ,entails a node G2 nodei-1 directly entailed by G2 nodei.
3. If there is entailment in both directions (paraphrase), perform step 2 in both directions.



The aim of the algorithm is to minimize the number of annotations (EDA calls in our case) needed to perform the merge.

This implementation of the module performs the procedure described above to merge the new fragment graph with each of the fragment graphs that are recognized within the given raw graph (i.e. were previously merged with the raw graph), while ensuring not to call EDA twice for the same pair of statements.

### Collapsed Graph Generator Module: class SimpleCollapsedGraphGenerator (eu.excitementproject.tl.composition.collapsedgraphgenerator)

This prototype implementation of the *CollapsedGraphGenerator* module produces a collapsed graph from a raw graph by performing the following simple steps:

1. Remove all the edges with confidence below a certain threshold (specified in the generator’s constructor). If no confidence threshold is specified, the average confidence is computed using all the edges in the graph, and edges with confidence below average are removed.
2. Recognize cycles and collapse all the nodes along each cycle’s path into a single EquivalenceClass node. The resulting graph is a transitive graph (with no transitivity violations).

### Node Matcher module: class NodeMatcherLongestOnly (eu.excitementproject.tl.composition.nodematcher)

This implementation of the *NodeMatcher* module compares an input fragment graph to an input entailment graph and tries to find the longest match: It starts with the complete statement, on which the fragment graph was built (i.e., the statement containing all modifiers), and tries to find a matching node in the entailment graph. If a match is found, it returns this matching node; otherwise it tries to match the strings on the next level of the fragment graph, i.e. the statements in which one modifier is missing. Again, if a match is found, the node(s) are returned, otherwise the process continues until the base statement (in which all modifiers are removed) is reached.

Internally, this implementation uses two additional (non-API) methods:

* *NodeMatch findMatchingNodesForMention(EntailmentUnitMention mentionToBeFound, EntailmentGraphRaw entailmentGraph)*

The purpose of this method is to find nodes in the entailment graph that match a particular mention. In this prototypical implementation, it goes through all the nodes in the entailment graph and compares the mentionToBeFound to all mentions associated to the nodes in the graph.

* + @param mentionToBeFound – the entailment unit mention to be found in the graph
  + @param entailmentGraph – the raw entailment graph against which the mention is matched
  + @return (NodeMatch) a node match for the mention to be found
* *double getNodeScore(EntailmentUnitMention mentionToBeFound, EntailmentUnit eu)*

This method computes a score expressing how well the mention matches the entailment graph node. In this prototypical implementation, it simply returns 1, if the mention exactly matches one of the mentions associated to the node, if not, it returns 0.

* + @param mentionToBeFound – the entailment unit mention to be found in the graph
  + @param eu – the entailment unit the mentionToBeFound is matched against
  + @return (double) score expressing how well the mention matches the node

### Category Annotator Module: class CategoryAnnotatorAllCats (eu.excitementproject.tl.composition.categoryannotator)

The *CategoryAnnotator* module adds category annotation to an input CAS, based on an input set of *NodeMatch*-es. This requires the combination of category information in the *NodeMatch*-es to category confidence scores. Each *NodeMatch* in the input set of *NodeMatch*-es holds exactly one *EntailmentUnitMention* *M* (found in the input CAS), which is associated to a list of *PerNodeScore*-s *P*. Each *PerNodeScore* in *P* refers to a tuple of an *EntailmentUnit* *E* (a node in a raw entailment graph) and a confidence score *p* denoting the confidence of *M* matching *E*.

In this implementation of the CategoryAnnotator module, category confidence scores are computed in the following way:

For each *P*, we first collect the category distribution on the node by retrieving the category of each of the *m* mentions associated to *P* and storing the sum of occurrences of this category. Let's refer to the category as *c*, and to the sum as *sum(c)*.

In the second step, we compute a score *s(c)* for each category occurring on the node using the following formula:

*s(c) = p \* sum(c) / m*

To compute the final category confidence, all scores for a particular category are summed up and, in the end, divided by the total number of *NodeMatch*-es.

## Implementation of Top Levels

### Use Case 1: class UseCaseOneRunnerPrototype (eu.excitementproject.tl.toplevel.usecaseonerunner)

The *UseCaseOneRunnerPrototype* provides an implementation of the *UseCaseOneRunner* interface. The interface’s *buildRawGraph* methods produce an *EntailmentGraphRaw* object from input CAS objects or input Interaction objects, by producing sets of *FragmentGraph*-s for each input interaction and merging them to produce an *EntailmentGraphRaw*. The interface’s *buildCollapsedGraph* methods produce an *EntailmentGraphCollapsed* object either directly from a raw graph or from input CAS objects / input Interaction objects, from which an *EntailmentGraphRaw* is built as described above and further collapsed (merging nodes into equivalence classes, collapsing multiple edges between the same pair of nodes into one edge, etc.), to produce the final output.

* Attributes:

To perform the transformations between the different types of graphs mentioned above, a few intermediate processing steps must be performed: fragment annotation, modifier annotation, generating fragment graphs from an input CAS object, merging of fragment graphs, and collapsing a raw graph. Each of these processing steps is encapsulated in an interface, which are attributes of the *UseCaseOneRunnerPrototype*, and are initialized by the *initInterfaces()* method. In addition to these interfaces, the class also has as attributes the EDA used in the merging step, and the LAP for producing the required input for the EDA. A summary of the attributes is presented below:

* + LAPAccess lap
  + EDABasic<?> eda
  + FragmentAnnotator fragAnot
  + ModifierAnnotator modAnot
  + FragmentGraphGenerator fragGen
  + GraphMerger graphMerger
  + CollapsedGraphGenerator collapseGraph
* Methods:

Besides implementing *UseCaseOneRunner*'s abstract method, the *UseCaseOneRunnerPrototype* implements the *initInterfaces()* method, which initializes the interfaces necessary for building the graphs. For now the method initializes a pre-specified set of interfaces:

* + FragmentAnnotator: *SentenceAsFragmentAnnotator* for adding (determined) fragment annotation of the input CAS
  + ModifierAnnotator: *AdvAsModifierAnnotator* for adding modifier annotations to each (determined) fragment
  + FragmentGraphGenerator: *FragmentGraphGeneratorFromCAS* for generating *FragmentGraph*-s for each (determined) fragment in the input CAS
  + GraphMerger: *AutomateWP2ProcedureGraphMerger* for merging *FragmentGraph*-s into an *EntailmentGraphRaw*
  + CollapsedGraphGenerator: *SimpleCollapsedGraphGenerator* to build an *EntailmentGraphCollapsed* from an *EntailmentGraphRaw*

Future versions of the class will make the initialization of these interfaces flexible, as the LAP and EDA already are (they are passed as arguments to the constructor).

### Use Case 2: class UseCaseTwoRunnerPrototype (eu.excitementproject.tl.toplevel.usecaseonerunner)

The UseCaseTwoRunnerPrototype provides an implementation of the *UseCaseTwoRunner* interface, which has one interface method *annotateCategories(JCas cas, EntailmentGraphRaw graph).* In the current implementation, this method annotates categories on the given input CAS based on an input entailment graph using the following module implementations:

* SentenceAsFragmentAnnotator: This module adds fragment annotation to the input CAS.
* AdvAsModifierAnnotator: This module adds modifier annotation to the input CAS.
* FragmentGraphGeneratorFromCAS: This module generates fragment graphs for the annotated fragments, using the modifier annotation.
* NodeMatcherLongestOnly: This module compares the created fragment graphs to the input entailment graph to find matching nodes.
* CategoryAnnotatorAllCats: This module adds category annotation to the input CAS by combining category information from the matching nodes.

## Implementation of Data Readers, and Other Utilities

### class CASUtils (eu.excitementproject.tl.laputils)

CASUtils is a class that contains a set of public methods. All of them are utility functions that are related to accessing CAS (JCas) data. The following summarizes the utilities provided by this static class.

* *JCas createNewInputCAS()*: this is the preferred method of generating a new JCas object.
* *void serializeToXmi(JCas, File)*: this method serializes the JCas into CAS-standard serialization format of XMI (file extension is XMI, where the content is XML representation of the CAS data structure).
* *void deserializeFromXmi(JCas, File)*: this method reads an XMI file and fills the content of the JCas with it.
* *annotateOneAssumedFragment(JCas, Region[]), annotateOneDeterminedFragment(JCas, Region[])*: the two utility methods annotate CAS data with fragment annotation. Each method gets a list of “regions” and annotates one fragment. The first method annotates *assumedFragment*, and the second annotates *determinedFragment*.
* *annotateOneModifier(JCas, Region[], ModifierAnnotation), annotateOneModifier(JCas, Region[])*: the two utility methods annotate CAS data with a modifier annotation. The difference between the two methods is the capability to handle “dependency”, where one modifier depends on the other. See [ModifierAnnotation](#_ModifierAnnotation_(eu.excitement.t), for explanation of the dependency between modifier annotations.
* *annotateCategories(JCas aJCas, Region r, String text, Map<String, Double> decisions)*: this method annotates the region r in the CAS with category annotation based on the decisions input (as part of use case 2).
* *void dumpCAS(JCas)*: this is a method that prints details of the JCas into standard output stream. The method is provided mostly for debugging.
* *void dumpAnnotationsInCAS(JCas aJCas, int annotType)*: this method is a "smarter" version of dumpCAS(), which gets the type of the annotation and only prints out that type within the input CAS. Just like dumpCAS(), this method is also provided for mostly debugging and checking.

### class InteractionReader (eu.excitementproject.tl.laputils)

For the moment, WP6 has one utility class that provides two readers for WP2 data. Both of them are static methods in the InteractionReader class.

* *List<Interaction> readInteractionXML(File)*: This method reads a WP2 interaction XML file, and converts it into a list of Interaction data type objects. Note that the Interaction data type is a simple Java class that represents non-annotated (no fragment, no modifiers) interaction based on strings. Each Interaction object can be easily converted to JCas input CAS by calling one of its member methods (*fillInputCAS(JCas),* or *JCas createAndFillInputCAS*())
* *readWP2FragGraphDump(File, File, JCas, String)*: This method reads WP2 fragment graph data. Note that WP2 fragment graph data has multiple files for one interaction. Interaction is given as a raw text file, and each fragment is given as one XML that denotes fragment graphs. To run this reader, one JCas, one XML and one raw text are to be provided.

# Plans for the Next Cycle

The Transduction Layer is currently in a prototypical state and can be viewed as a “proof of concept.” So far, it has only been tested on a small amount of sample data and the module implementations can be seen as examples illustrating the purpose of each module. In order to show that it can actually be used for the different industrial use cases, we plan to extend the transduction layer in the following ways.

## Provide an Experimentation Environment

First of all, it is essential to create an environment that allows us to test and evaluate the performance of our modules on the WP2 datasets. These datasets can be used for the following:

1. Testing the Transduction Layer modules and evaluating the performance of different implementations on these datasets.
2. Creating training data to train the EDAs provided by the EOP.

For evaluation and analysis of the generated entailment graphs, visualization may be essential. In the current prototype, we already use one external visualization package. For visualizing larger graphs, however, we will analyze what exactly needs to be visualized and which packages would best serve our purposes. We plan to make this decision in synergy with the industrial partners, in order to make the visualizations suitable for their needs as well.

## Provide More Sophisticated Module Implementations

The current implementations of the modules are prototypical. The implementation of more sophisticated modules will be based on the evaluation over the WP2 data. Currently, we plan the following improvements.

### InteractionReader

The reader code (in the InteractionReader class) currently cannot read  
"non-continuous" fragments (which are common in the WP2 data). If the reader meets such a case, it raises an exception and gives up. Writing code for this is not trivial because  
WP2 data does not have fragment annotation and some type of alignment has to be done. Note that CASUtils does provide a helper function to annotate CAS with non-continuous fragments, and we can easily make non-continuous fragments. The limit is only on the WP2 data reader.

The same limitation currently holds for modifiers: Non-continuous modifiers  
are not covered. However, non-continuous modifiers are very rare in  
the WP2 data.

### Decomposition

In the decomposition part, our focus in the next cycle will be on fragment and modifier annotation. For both tasks, we aim to first come up with a proper definition of the task (based on related research). For modifier annotation, we plan to examine the field of “sentence compression”, which seems related to our task of recognizing modifiers. A starting point here would be to have a look at Mirella Lapata's line of work[[1]](#footnote-1) to see if we can apply some of their approaches, or just get some inspiration from their work.

### Composition Use Case 1

In the composition part, we will work in two following directions:

1. Improving the *merge graph* procedure
   * Evaluating and analyzing the limitations of the WP2 merging procedure when using automatic EDA-based entailment decisions, rather than manual annotation as done in WP2.
   * Following the results of this analysis, adjusting the algorithm to cope with inexact entailment decisions, or developing alternative merge procedures based on the insights from the analysis.
2. Developing a sound *collapse graph* procedure
   * For this direction of research we plan to examine the applicability of the Global Graph Structure Optimization algorithm of Berant et.al[[2]](#footnote-2), which is being implemented as part of WP5. The algorithm was suggested for learning entailment relations between predicates and is based on global optimization of an entailment graph structure in order to avoid transitivity violation.

We will give attention to efficiency issues. For example, thorough technical consideration will be needed on saving and reusing LAP annotation on the graphs, especially related to multiple EDAs.

### Composition Use Case 2

In the current, prototypical implementation, the Node Matcher uses an exact match approach, i.e., it only returns nodes that contain a mention that exactly matches the input string. When working with the industrial data, the matching step needs to become more tolerant. This may include tolerance concerning spelling differences, but also tolerance concerning morphological (or even syntactic) variation. The definition of the kind and degree of “tolerance” is a matter of research.

Concerning the Category Annotator module, more sophisticated algorithms for combining category information on a graph node into a category confidence score may be explored. For this, we will investigate related work and run experiments using WP2 data.

1. Trevor Cohn and Mirella Lapata: [Sentence Compression as Tree Transduction](http://homepages.inf.ed.ac.uk/mlap/Papers/jair09.pdf). Journal of Artificial Intelligence Research 34(1) pp. 637-674 (2009) [↑](#footnote-ref-1)
2. Jonathan Berant, Ido Dagan and Jacob Goldberger Learning Entailment Relations by Global Graph Structure Optimization. Long paper in The Journal of Computational Linguistics 38(1) pp. 73-111 (2012) [↑](#footnote-ref-2)