

From texts to structured data: Building knowledge graphs through Computer-Assisted Semantic Text Modelling (CASTEMO)

This book documents:

- a data collection workflow of Computer-Assisted Semantic Text Modelling (CASTEMO);
- the structure of CASTEMO knowledge graphs;
- the research environment implementing this workflow - InkVisitor.

- [About the book](#)
- [Acknowledgements](#)
- [List of abbreviations](#)
- [Why knowledge graphs?](#)
- [Entities](#)
 - [Actions](#)
 - [Concepts](#)

- [Attributes of entities](#)
 - [Statements](#)
- [Properties](#)
 - [Properties overview](#)
- [Relations](#)
 - [Relations overview](#)
 - [Superclass \(SCL\)](#)
 - [Superordinate Entity \(SOE\)](#)
 - [Classification \(CLA\)](#)
 - [Identification \(IDE\)](#)
 - [Synonym \(SYN\)](#)
 - [Antonym \(ANT\)](#)
 - [Holonym \(HOL\)](#)
 - [Property Reciprocal \(PRR\)](#)
 - [Action/Event Equivalent \(AEE\)](#)
 - [Implication](#)
 - [Subject/Actant1 Reciprocal \(SAR\)](#)
 - [Actant semantics: Subject Semantics \(SUS\), Actant 1 Semantics \(A1S\), and Actant 2 Semantics \(A2S\)](#)
 - [Related \(REL\)](#)
- [References](#)
- [How best collect CASTEMO data?](#)
 - [Describe your data collection choices](#)
 - ["Same as above": Referencing information content in CASTEMO knowledge graphs](#)
- [Full-text annotation](#)
 - [Import a full-text document and start the annotation work](#)
- [Querying CASTEMO knowledge graphs](#)
 - [Finding inconsistent and invalid data](#)

- [Querying CASTEMO knowledge graphs in Neo4j](#)
 - [Querying with relations](#)
- [Data import](#)
 - [Parsing instructions for the import of tabular data](#)
- [InkVisitor installation on the server](#)
 - [How to deploy your own instance of InkVisitor](#)

About the book

Authors: David Zbírál, Robert L. J. Shaw, Tomáš Hampejs, and Adam Mertel.

You can contact David Zbírál at david.zbiral@mail.muni.cz.

Acknowledgements

The CASTEMO data collection workflow and the [InkVisitor](#) research environment were developed as part of the [Dissident Networks Project \(DISSINET\)](#), based at the Centre for the Digital Research of Religion, Department for the Study of Religions, Faculty of Arts, Masaryk University.

We gratefully acknowledge the generous financial support of the following institutions and funding schemes:

- **European Regional Development Fund** (grant agreement No. CZ.02.01.01/00/22_008/0004595, project “Beyond Security: Role of Conflict in Resilience-Building”), 10/2023-6/2028.
- **European Research Council (ERC)** under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 101000442), 9/2021-9/2023.
- **Masaryk University, Faculty of Arts, Dean's Grant of Excellence in Research**, under the project "InkVisitor Development: Towards a Project-Neutral Open-Source Research Application for the Collection of Structured Relational Data from Texts", 6/2022-6/2023.

List of abbreviations

Abbreviation	Meaning
A	Action type
B	Living Being
C	Concept
E	Event
G	Group
L	Location
O	Object (i.e. physical thing)
P	Person
R	Resource
S	Statement
T	Territory
prop	Property
metaprop	Metalevel Property

Why knowledge graphs?

Knowledge graphs are flexible data structures which store data as nodes and ties (edges). Knowledge graphs as a general data structure are *not* limited to any particular methodology of data analysis (such as network analysis); rather, they allow the **modelling of textual data** in ways very close to original sources, enable various **reasoning capabilities** through logical inferences, and allow a **flexible schema**, which evolves with the progress of data collection and research.

CASTEMO (Computer-Assisted Semantic Text Modelling) is a **workflow for the collection of data from textual sources**. Its goal is to produce queryable, well-structured, **multi-lingual, research-oriented knowledge graphs** for **long-term and versatile use**, rather than for a single use in one publication. It is thus well-adapted, for instance, to **historical research** and projects which combine **qualitative and quantitative methodologies**.

The **CASTEMO data model** contains some pre-defined, and a large variety of flexible, user-defined features. It is based on **11 entity types (SPECTRABLOG - Statements, Persons, Events, Concepts, Territories / Texts, Resources, Actions, Living Beings, Locations, Physical Objects, and Groups)** and **three kinds of connection between entities: Properties, Relations, and References**.

Properties are flexible and extensible structures composed of an origin (the entity to which the Property is being attached), property type, and property value, and read with a “has” logic: e.g.: P current U.S. president – PROP – C area of authority – L United States of America”.

Relations, of which there are seventeen types, serve to model core ontological (e.g. Classification, Identification) and semantic (e.g. Synonymy, Superclass) connections.

References serve to relate knowledge to a specific Resource from which it has been derived.

On the whole, **knowledge graphs provide the best way of structuring complex data in ways close to the original expression in the sources**, keeping the analytical layers safely stored but neatly separated from the textual layer.

Entities

This chapter describes the different entity types of the CASTEMO data model, and their recommended usage.

Actions

Actions (or more fully, Action types) represent **individual semantically disambiguated verbs**. They are **lemma-meaning units**, i.e. one meaning of a specific lemma corresponds to one Action. Thus, we can have many Actions labelled "to see": one for "be able of sight, not blind", one for "perceive with sight", one for "go to meet somebody".

Three valencies: Entity type valency, grammatical valency, and semantic valency

Introduction

Actions acquire three kinds of valencies for any actant slot (subject, object 1, object 2; the data model is potentially extensible further, beyond trivalent verbs):

1. **entity type valency**, which defines which entity type (Person, Concept, etc.) is allowed in the given actant slot;
2. **morphosyntactic valency**, which is a free text field defining the **prepositions and grammatical cases**, but uses a formalized notation (grammatical cases are noted with numbers, prepositions are in quote marks "", alternative is marked with a pipe "|"); and
3. **semantic valency**, i.e. what kind of role the entity occupying the given actant slot has by implication (e.g., the subject of the Action "to travel" would have the semantic valency C "traveller").

The main benefits from valencies are that they:

1. guide users in their **choice of the correct Action** (or creating a new one if none among the existing fits the meaning and syntactic structure);
2. help users with **validity of data in actant slots**;
3. allow InkVisitor to deploy **data validation features**;
4. facilitate **machine understanding of text**, allowing semantic disambiguation of verbs based on their morphosyntactic valency (recognized by dependency parsing), and optionally, entity type valency (recognized e.g. through named entity recognition).

Morphosyntactic valency notation for Latin

In the field marking morphosyntactic valency, we use the following **abbreviations and signs**:

- **Numbers 1-6**: cases. E.g. "1" means nominative, "6" means ablative.
- **Pipe sign ("|")**: denotes the logical "OR", i.e. marks alternative morphosyntactic valencies.

- **Plus sign (" + ")**: denotes concatenation, e.g. "de" + 6 means: "with preposition *de* and ablative case".
- **Words in quote marks ""**: denote the actual words used in this valency, e.g. prepositions in this valency.
- **inf**: infinitive.
- **4inf**: accusative with infinitive.

E.g., `4 | 4inf | "quod"` means that in this actant slot, this verb can only take either an accusative, or a sentence rendered as accusative with infinitive, or a clause starting with "quod".

Recommended standards for a finalized (`approved`) action

Before assigning an Action the `approved` status, it should meet the following standards:

- Its **meaning is described** in the `detail` field. (You will benefit from the use of printed or online dictionaries or LLMs.)
- It has the **Action/Event Equivalent relation filled in** with a Concept which has its meaning defined in its own "detail" field.
- It has **full information on the three valencies** for each actant slot (including the explicit declaration of empty in the entity type valency, if no entity is allowed in that slot).
- It has a **reference to an external lemma collection ID** (in DISSINET, we use the LiLa Lemma Collection).
- If you **have found a corresponding meaning among WordNet synsets**:
 - It has a **Reference to the corresponding WordNet synset**.
 - Its definition in the `detail` field **takes the WordNet definition into account**.
- If you **haven't found a corresponding meaning among WordNet synsets**:
 - You have **defined the meaning** yourself or based on dictionaries.
 - If there is any **synset in WordNet which is a superclass** of this (more specific) meaning, then **an Action corresponding to the WordNet meaning is created** (if Latin WordNet has it, then in Latin; if not, then in English), **described, has a Reference to the WordNet synset, and it forms the Superclass** of this more specific Action you are working on.
- There is **no remaining error message from InkVisitor validation**.
- All of this has been **checked**, i.e. it is not just a first draft of the Action that you still plan to come back to.

For something to be aligned with a synset definition in WordNet, it is *not* required that you accept its hypernyms or synonyms, just the definition needs to match.

Recommended linkage to external lemma and meaning banks

- **Link each Action through a Reference to at least one external lemma bank.** A major lemma bank is still **WordNet for the given language**. For Latin, we use the [LiLa Lemma Collection](#) in DISSINET.

- **Link each Action through a Reference to at least one external bank of meanings/senses.** A major meaning bank is still **WordNet synsets**.
- Such linkages are important for the **interoperability of your data**, and giving it meaning curated by bigger projects and infrastructures.
- **Reference** is a pair composed of a Resource entity representing the given resource, and its part (typically unique identifier).
- If the lemma or meaning is **not found** in the lemma or meaning bank you are normally using, it is useful to know that you checked: in such a case, **add the Reference, but put Value "NA" as the Reference part**.
- Also **DISSINET Database (DDB)** and **MedHate database (MDB)** are providers of **identifiers (UUIDs) that you can link to**. If upon its creation you request to have your CASTEMO database pre-populated with some entities from another deploy of InkVisitor, the UUIDs will be the same and the references will already be there.

Concepts

Concepts represent, alongside Action types, another **generic entity type**, which holds the data semantically together. Concepts are **lemma-meaning units**. That is, for **any distinct lemma and meaning, you create a new Concept**. This proliferation of Concepts is made manageable through various [Relations](#), such as **synonym, superclass** (~ hypernym, genus proximum) etc., which place any Concept in a robust web of semantic relations.

One major function of Concepts is to serve as **Property Type** in [Properties](#), which is a flexible yet reliable way of creating connections between entities.

CASTEMO knowledge graphs are **multi-lingual**; thus, Concepts from different languages can coexist, but are defined by semantic relations.

While CASTEMO knowledge graphs are multi-lingual, we recommend choosing **one analytical language** for higher levels of the conceptual taxonomies.

Attributes of entities

Any entity type has some **internal Attributes**, which allow to characterize the entity. The InkVisitor interface guides users as to what attributes are expected for a given entity type.

A first and obvious Attribute is **label**, that is, the name of the entity. The name can change; it is the identifier (UUID) rather than the label which holds the semantic identity. However, **label changes** need to be done with consideration, as they influence the uses of this entity in already existing data.

The **detail** Attribute should be used to **define** the entity. This is especially important for Actions and Concepts, which keep the data semantically together.

Another attribute used in all entities is **label language**, which defines in what language the label is written.

Languages not available in the interface (including old development phases of modern languages) can be added upon request.

To further extend the possibilities of finding an entity rather than create duplicates e.g. for mere **orthographic variants** (e.g. democratisation vs. democratization), there are **alternative labels**, which means other labels than the main one, highlighted in label.

Alternative labels should *not* be used for labels in another language (create the entity in that language and link it with a SYN Relation to this one instead).

In Concepts and Actions, alternative labels should *not* be used for different lemmas.

Technically, the main label and alternative labels are one single field; the first position in this field is held by the main label. This is how the application ensures non-duplication between main label and alternative labels.

Apart from such internal Attributes, any entity can enter in **three types of connections to different entities**: Properties, Relations, and References.

Statements

Structure and purpose

Statements model the syntactic structure and semantics of clauses. They have a **quadruple structure** with **action slot** and **three actant slots: subject, actant1, and actant2**. The semantic core is the **action** slot, which holds the **predicate** of the clause, and is linked to actants as defined by the **syntactic valency** of the Action type used, up to trivalent verbs (for instance, 'Peter received a gift from Elisabeth'). Any Action Type can define through its **syntactic valency** that an actant is required, optional, or forbidden (required empty).

While InkVisitor can be used for classical entity-relationship modelling done in any database, the CASTEMO data collection workflow is specific in its focus on **modelling textual clauses** through statements. In this sense, it is a **statement-based data collection workflow**. This allows texts to be comprehensively modelled, either selectively (based on what is relevant for specific research) or in entirety. CASTEMO is intended to **keep the order of appearance and contextual embeddedness of information** in the text.

Action attributes

Actant attributes

Pseudo-actant

Properties

This chapter explains a vital kind of relation in the CASTEMO data model: properties, which serve to attach attributes to entities, and relate entities with one another.

Properties overview

Definition

Property

Property is the most fundamental and the most versatile relation in CASTEMO knowledge graphs which connects **entities of different kinds**.

Any property has:

1. an **origin**, i.e. the entity to which it is attached,
2. a **property type**, and
3. a **property value**.

Properties serve for modelling:

- **Adjectives, adjectival expressions, possessives, and appositions** (e.g., qualities and inter-entity relations).
- **Adverbs and adverbial expressions** (e.g., time, space, and manner of action).

CASTEMO knowledge graphs give priority to the **data-collection perspective**, and attempt at creating a **general** framework for modelling information in the textual resources. That's why it does not have a comprehensive list of predefined properties but allows coders to fill them during data collection flexibly and dynamically.

Any entity can have **any number of properties**.

If an entity has **more qualities of the same kind**, those will be multiple properties with the same property type (e.g. if something is yellow and blue, it will have one property for yellow and the other for blue colour).

In CASTEMO knowledge graphs, a Property is always read with the verb "has".

E.g.:

O John's hat - [has] - C colour - C black

P Anna - [has] - C son - P Peter

There are two kinds of properties, which have the same structure but differ in the context from which they are attached to entities.

In-statement Properties

In-statement Properties are attached to entities **contextually**, from within a Statement - i.e. they are attached to entities, used as **actants** of a Statement (in the case of adjectival expressions and appositions) or **actions** (in the case of adverbial expressions). Thanks to the position of Statements in CASTEMO knowledge graphs, such Properties are included in a Territory hierarchy from which they stem, and usually accompanied with a either an anchor in a full-text document, or a Reference to the Resource which served for compiling a given Statement.

Meta-level Properties

Meta-level Properties (or, in short, **Metaproperties**), are properties attached **globally** rather than contextually to an entity, and represent general knowledge which applies to all uses of the entity independently from context. For instance:

```
P Martha Wood - [has] - C sex - C female
```

Sometimes, **one level is not enough** to model a property. E.g., if Elizabeth II was Queen of the United Kingdom from 1952 to 2022, it is important to declare both the role and its temporal span, e.g. in the following way:

```
P Elizabeth II - [has] - C office - C Queen of the United Kingdom
```

```
PROP - C TRP yyyy-mm-dd: started exactly - V 1952
```

```
PROP - C TRP yyyy-mm-dd: ended exactly - V 2022
```


It is not Elizabeth who has this time span, but her office as queen of the United Kingdom. Thus, CASTEMO knowledge graphs have **second-level properties to be able to attach properties to properties**, and they have even **third-order properties** to attach properties to properties of properties.








Technical specification

The data model of CASTEMO knowledge graphs is **technically** defined in the InkVisitor **typescript *.ts files**, which directly correspond to the JSON structure of the collected data in the corresponding database.

Each **entity** (e.g. Action) has its own **ts file** (e.g. *action.ts*); see the [GitHub repository](#):

dev **InkVisitor** / packages / shared / types /

 adammertel and jancimertel 1494 search entities by territoryid (#1503) ...

..	
 actant.ts	use namespace to divide enums in shared directory
 action.ts	add missing interfaces for entities' data subclasses
 audit.ts	Fix/constructors (#791)
 being.ts	Add new entity class - living being
 concept.ts	use namespace to divide enums in shared directory
 entity-tooltip.ts	Response detail relations (#1413)
 entity.ts	use namespace to divide enums in shared directory

In the sense of object-oriented design, each specific entity (e.g. location, person) inherits the definition of the **IEntity interface**, which is in the *entity.ts* file. The **IEntity** is defined as follows:

20 lines (19 sloc) | 453 Bytes

```
1  import { IProp } from ".";
2  import { EntityEnums } from "../enums";
3  import { IReference } from "../reference";
4
5  export interface IEntity {
6      id: string;
7      legacyId?: string;
8      class: EntityEnums.Class;
9      status: EntityEnums.Status;
10     data: any;
11     label: string;
12     detail: string;
13     language: EntityEnums.Language;
14     notes: string[];
15     props: IProp[];
16     references: IReference[];
17     isTemplate?: boolean;
18     usedTemplate?: string;
19     templateData?: object;
20 }
```

Example: location entity "Argentina"

Let's have a location entity with the label "Argentina".

In the **InkVisitor GUI**, the core entity is displayed like this:

The screenshot shows the InkVisitor GUI for an entity named 'Argentina'. At the top, there's a green tab labeled 'L' and 'Argentina' with icons for delete, copy, and paste. Below this, the ID is 'b14a5804-b916-4a1d-b9a0-bf50bec1f7e7' with a copy icon. The Entity Type is a dropdown menu set to 'Location'. The Apply Template field shows 'select template'. The Legacy ID is 'L0001_R0007'. The Label field contains 'Argentina'. The Detail field is empty. The Status field has four buttons: 'pending', 'approved' (highlighted in blue), 'discouraged', and 'warning'. The Label language is a dropdown menu set to 'Latin'. The Logical Type field has four buttons: 'definite' (highlighted in blue), 'indefinite', 'hypothetical', and 'generic'.

...

In the **InkVisitor GUI**, the properties which hold its geospatial localisation are displayed like this:

The screenshot shows the 'Meta properties' section. It contains two rows of property records. The first row has a green 'C' icon, the text 'coordinates (lat, long)', a delete icon, a value field '48.58188; 7.75104' with a delete icon, and a settings icon, a delete icon, a '+p' icon, and an 'a' icon. The second row has a green 'C' icon, the text 'localisation precision', a delete icon, a value field 'precise' with a delete icon, and the same settings, delete, '+p', and 'a' icons. At the bottom left, there is a dark blue button with a white '+' icon and the text 'create new meta property'.

There is **one property record** with type-value pair, i.e. **Property Type** "coordinates (lat; long)" (a Concept entity, an entity object of type C, which is taken is from the predefined ontology) and the **Property Value**, in this case a **Value object** with the string holding the decimal geographic coordinate pair. To this first-level property is attached a second-level property defining "localisation precision" of the stated coordinates, where both values of the type-value pair are filled with C entities.

For deeper conceptual understanding, the **JSON structure** can be more revealing, this is a full entity display:

```

{
  "root": {
    "class": "L",
    "data": {
      "detail": "",
      "entities": {
        "id": "b14a5804-b916-4a1d-b9a0-bf50bec1f7e7",
        "label": "Argentina",
        "language": "lat",
        "legacyId": "L0001_R0007",
        "notes": [
          "precise"
        ],
        "props": [
          "coordinates (lat, long)"
        ],
        "references": [
          "localisation precision"
        ],
        "relations": [
          "48.58188; 7.75104"
        ]
      }
    }
  }
}

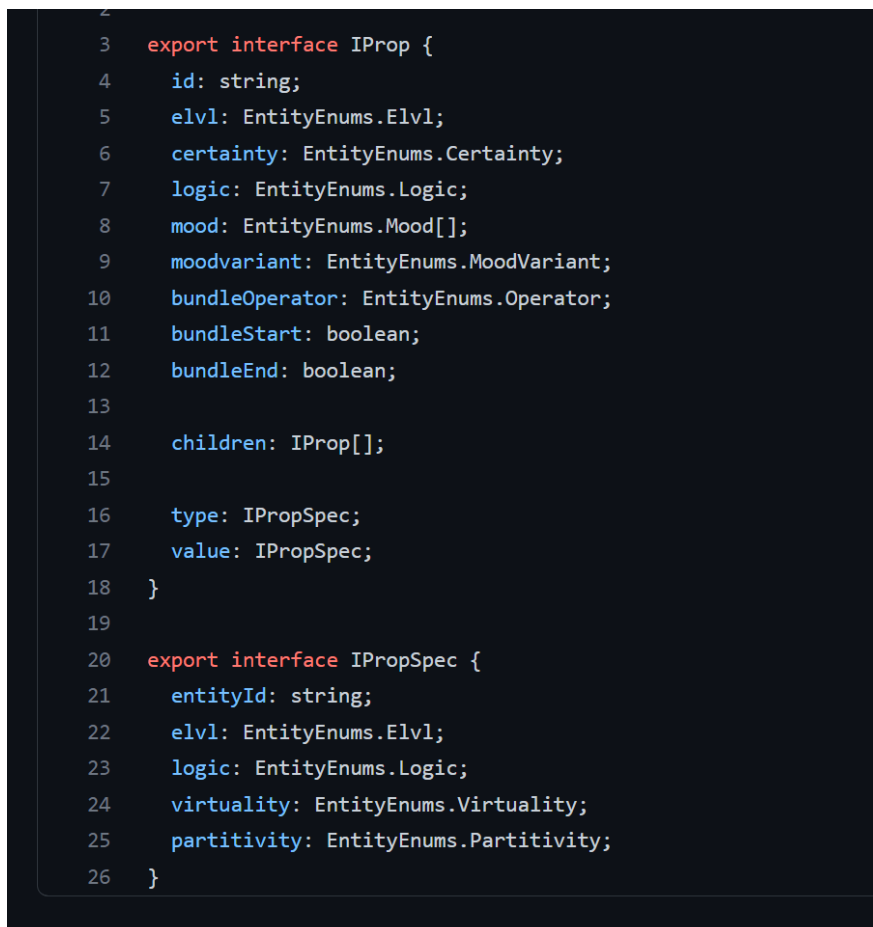
```

The entity has class "L", i.e. it is an entity of type "location", and it has direct attributes, like *label* with the value "Argentina" and array attributes like *props*.

It has one property object in the *props* attribute. The main information is held in **type** and **value** attributes. And this property object also has one **second-level property** in the attribute *children*.



The property object is defined by the class *IProp*, see [prop.ts file](#).



Second-level properties

In the Argentina example above, the coordinates property record has one property object which extends the information.

```
▼ "children" : [ 1 item
  ▶ 0 : { . . . } 12 items
]
```

This object is a normal instance of the IProp class. This means that its primary information is in type and value attributes. These hold *IPropSpec* objects, which refer to two concept entities, e.g. here, "localisation precision" and "precise".

```
▼ "children" : [ 1 item
  ▼ 0 : { 12 items
    "bundleEnd" : false
    "bundleOperator" : "a"
    "bundleStart" : false
    "certainty" : "0"
    ▶ "children" : [] 0 items
    "elvl" : "3"
    "id" : "5a206e04-f7f6-4444-9575-d3d77599271e"
    "logic" : "1"
    ▶ "mood" : [ . . . ] 1 item
    "moodvariant" : "1"
    ▶ "type" : { . . . } 5 items
    ▶ "value" : { . . . } 5 items
  }
]
```

Relations

Some core semantic and ontological relations between entities are highlighted in the CASTEMO data model.

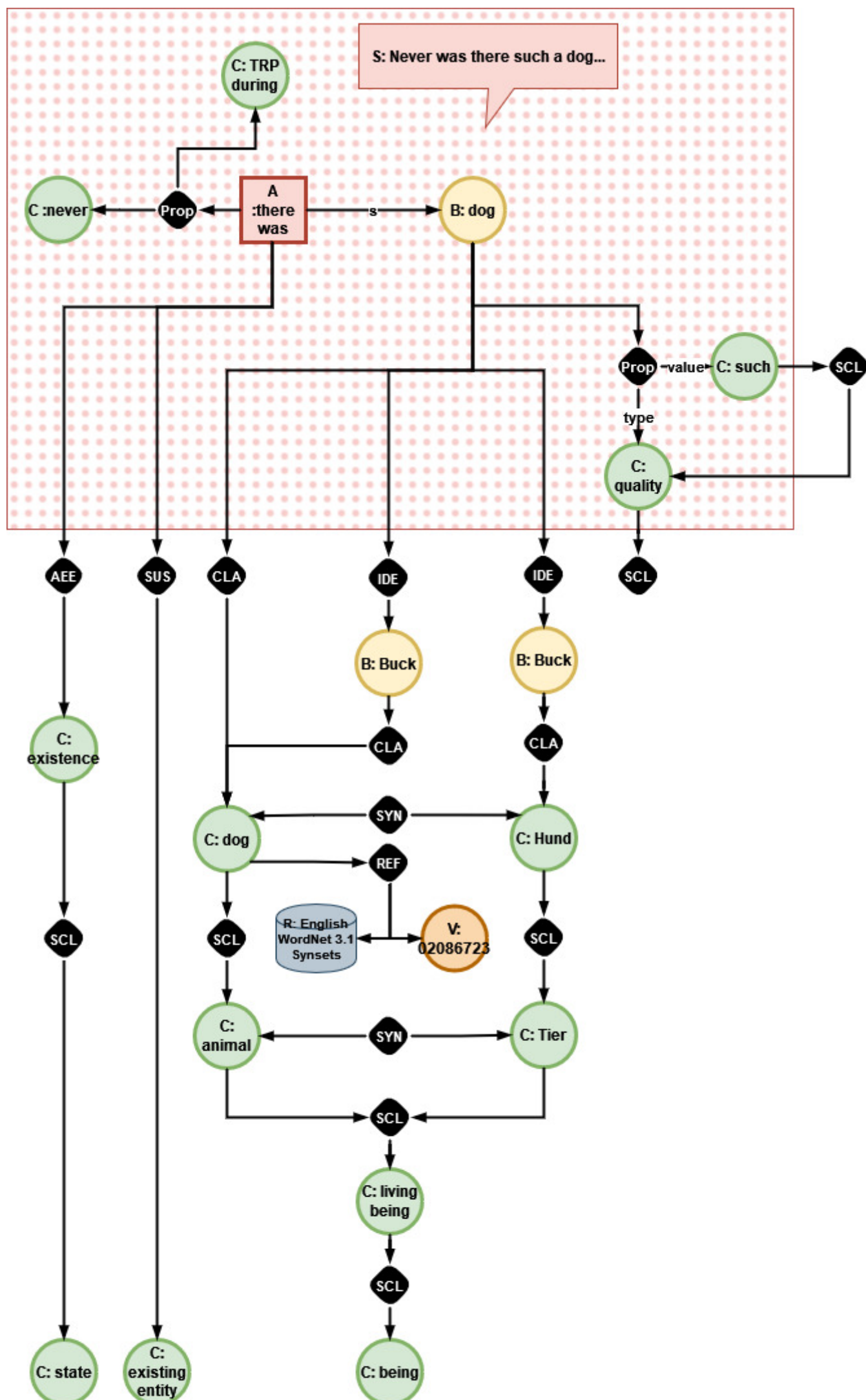
Relations overview

Some core semantic and ontological relations between entities are highlighted in the CASTEMO data model. Their reserved nature - in ways unavailable for specific Property Types, which depend on specific UUIDs - ensures their compatibility across CASTEMO research databases (and InkVisitor deploys). The following table gives an overview of all the relations, the entity types which allow that relation, a brief definition, and an example.

Relation name	Abbreviation	Name in JSON code	Inverse relation name	Allowed entity combinations	Detail	Example
Superclass	SCL	Superclass	Subclasses	A-A, C-C	Superordinate term (hypernym).	C apple -> C fruit A walk -> A move
Superordinate Entity	SOE	Superordinate Entity	Subordinate Entities	L-L, O-O, G-G, E-E, S-S, E-S, S-E, R-R (under discussion: T-T)	Ontological superset.	L Milan -> L Italy O the handle of this door - O this door
Synonym	SYN	Synonym	Synonym	A-A, C-C	Synonym both within a language and across languages (equivalent).	C funny - C strange
Antonym	ANT	Antonym	Antonym	A-A, C-C	Opposite term.	C good - C bad
Property Reciprocal	PRR	PropertyReciprocal	PropertyReciprocal	C-C	The concept that the property reciprocates if read the other way.	C parent <-> C child

Subject/Actant1 Reciprocal	SAR	SubjectActant1Reciprocal	SubjectActant1Reciprocal	A-A	The action that the actant1 gives back to subject.	A hear (from sb - about st) <-> A tell (sb - about st)
Subject Semantics	SUS	SubjectSemantics	Used as Subject semantics	A-C	Semantics of the subject (actant 0) slot.	A talk (to sb - about st) -> C speaker
Actant1 Semantics	A1S	Actant1Semantics	Used as Actant 1 Semantics	A-C	Semantics of the actant 1 (object 1) slot.	A talk (to sb - about st) -> C listener
Actant2 Semantics	A2S	Actant2Semantics	Used as Actant 2 semantics	A-C	Semantics of the actant 2 (object 2) slot.	A talk (to sb - about st) -> C speech content
Action/Event Equivalent	AEE	ActionEventEquivalent	Action equivalent	A-C	What is this action in the world of nouns?	A baptize - C baptism
Related	REL	Related	N/A	SPECTRAL OGB- SPECTRAL OGB	Free relation to connect things not connected by more specific semantic relations.	C tea -> C afternoon
Classification	CLA	Classification	Instances	PLOGESTR B-C	Relations of specific entities to the concepts they are the instance of.	O this apple -> C apple

Identificati on	IDE	Identification	Identification	PLOGESTR B- PLOGESTR B	Declaration of ontological identity between things.	L Italie (French) <- > L Italy (English)
Holonym	HOL	Holonym	Meronyms	C-C	Relation of a part to its whole.	C gate of a monastery -> C monastery
Implication	IMP	Implication	Used as Implication	A-A	Action implied by this action.	A dine (with sb) - > A be in the company (of sb)
In- statement Classificati on			In-statement instances	PLOGESTR B-C	Same as Classificati on, but done from within a Statement.	
In- statement Identificati on			In-statement Identification	PLOGESTR B- PLOGESTR B	Same as Identificati on, but done from within a Statement.	



Superclass (SCL)

Superclass (SCL) is a semantic relation which relates an Action to one or more Actions, or a Concept to one or more Concepts. It denotes **hypernym (genus proximum)**, i.e. the more generic conceptual class to which an Action or Concept belongs.

Example: `C apple - SCL - C fruit`.

Do not confuse Superclass with Classification. Classification serves to relate specific entities (PLOGESTRB) to its Class (C). All of these specific entities are then Instances of this Class.

Superordinate Entity (SOE)

Superordinate Entity (SOE) is a Relation which connects a subordinate entity to an entity in which it is fully contained.

Example: `L Milan - SOL - L Italy`.

In CASTEMO, only the direction from the subordinate to the superordinate entity is being stored. However, subordinate entities are displayed in InkVisitor to give an overview of the data.

Classification (CLA)

Classification (CLA) is a Relation between a specific PLOGESTRB entity and the class (Concept) to which it belongs.

Example: `0 this apple - CLA - C apple`; `0 another apple - CLA - C apple`.

Classification is **vital for querying the database**: it allows to find specific instances of a generic Concept (e.g., all apples) but also, through the Superclass tree of Concepts, more general classes (e.g., any instances of fruit).

Do not confuse Classification with Superclass.

Identification (IDE)

Identification (IDE) serves to declare the **identity** between PLOGESTRB entities, both **within an entity type** (e.g. `P Rocket Man - IDE - P Elton John`) and **across entity types** (`L the boat on which we got married - IDE - O this same boat that we then sold`).

Not unlike Synonymy, it helps preserving the original expressions used in the modelled Resources, while still keeping them identical. However, Synonymy pertains to Actions and Concepts and is a *semantic* relation, while Identification pertains to entities of this world, and is an *ontological* relation.

Unlike all other Relations, Identification can be done with a **Certainty level**, i.e. the identification can be done at a lower Certainty level, e.g. "probable", "possible", or even "false", which is a negative declaration that the two entities are *certainly not* identical, which can sometimes be useful if some resource has made this (erroneous) identification.

Any queries of entities must take into account the Identifications of the entity (usually including the definition of threshold certainty levels), if they want to include also all of its identicates in search results.

Entities with which an entity is identical are called **identicates**.

Synonym (SYN)

The CASTEMO data model recommends a **strong understanding of synonymy**. For two [lexemes](#) to be related with the Synonym Relation, they need to have the **same Superclass** (if there are more superclasses, they need to share all of them). Also, if they are related to a meaning bank, **the meaning ID in the relevant meaning bank needs to be the same**.

This strong understanding allows to consider synonymy as **transitive**: if A is synonym of B and B is synonym of C, then A is synonym of C. (In semantic network terms, this would be described as *triadic closure*.)

The Synonym Relation is also used for **cross-language equivalence**, e.g. C dog (English) - SYN
- C Hund (German).

The Synonym relation helps to **extend database queries** and also include the Synonyms of Concepts and Actions queried. Through Concepts used in Classification, synonymy can also help build **sets of entities for analysis** (e.g. including all Living Beings classified either as "dog (English)" or as "hound (English)" or as "Hund (German)").

Antonym (ANT)

The CASTEMO data model recommends a **strong understanding of antonymy**, i.e. one which to some degree applies also to the Superclasses of both lexemes.

Through **Concepts used for Classification**, antonymy helps to build **contrasting sets of entities for analysis**. Through **antonymous Actions**, it helps to build contrasting sets of Events.

Holonym (HOL)

Holonym (HOL) Relation denotes the relation between a Concept representing a part of something to a Concept representing a whole.

Example: `C door of a house - HOL - C house`.

The reverse relation is called meronymy.

In CASTEMO knowledge graphs, we only store the relation from the part to the whole, not the other way around, but the reverse direction is displayed in InkVisitor.

This ontological relation between Concepts is analogical to what the SOE Relation is between PLOGESTRB entities.

Property Reciprocal (PRR)

Property Reciprocal (PRR) is a Relation connecting two Concepts which can feature as a Property Type. In a Property, composed of Origin, Property Type, and Property Value, it defines what Property Type the Property Value would give back to Origin if the Property would be stored the other way. For instance, the PRR of C "mother" will be "child", because if `P Bernard - has - C mother - P Anna`, then `P Anna - has - C child - P Bernard`. Similarly, the PRR of C "occupation" will be "occupation holder", because if `P Bernard - has - C occupation - C baker`, then `C baker - has - C occupation holder - P Bernard`.

Property Reciprocal serves to symmetrize the Properties in a knowledge graph derived from the Collected Data Database by enriching them with the Property sent the other way, thus enabling to **query CASTEMO data more easily** and avoid missing properties which happen to be encoded the other way around.

Action/Event Equivalent (AEE)

The **Action/Event Equivalent Relation (AEE)** connects always one Action to one Concept, and it serves to translate between the world of verbs and the world of nouns. For instance, A "to baptize" would have the AEE C "baptism".

Since any specific Event which happens to be a baptism should have the Classification C "baptism", all specific "baptism" Events and all Statements with the Action "to baptize" used in the action slot can be **queried together** in the database, as long as they have the AEE and CLA properly set.

Just as importantly, AEE is **a way for verbs to acquire coherent Superclasses**. While the Superclass Relation is allowed for Actions, most verbs do not have manifest Superclasses. For instance, A "to baptize" does not have any Superclass such as *"to ritualize", while C "baptism" can acquire quite a straightforward Superclass tree (e.g. C "Christian ritual", which in turn will have the Superclass C "ritual"). These Superclasses propagated to Actions from Concepts through AEE are shown under Actions in InkVisitor, thereby helping coders to decide whether an Action fits the specific context. They are of course vital for **analytical queries** which aim at retrieving, for instance, all rituals, be they expressed through verbs (i.e., through an Action) or nouns (i.e., as Events).

Implication

Implication (IMP) is a Relation which connects an Action to one or more other Actions. It denotes an action which, by implication, must have happened because it is logically implied by the original action. For instance, `A travelled (with sb) - IMP - A was in the company (of sb)`.

Implication allows to **enrich data** in the knowledge graph with actions which, while not explicitly denoted by the verbs used, happened by implication. This in turn helps to extend the query results in cases where the Action's Superclass or Action/Event Equivalent did not necessarily cover the implied action.

Depending on the strictness of use of IMP and the research purpose, IMP can be used to produce only the first-level implication directly appended to the Action, or to follow the implication chain even further (e.g. if action A implies action B, and action B implies action C, then we may decide to also make action A imply action C upon data projection).

In linguistics, this relation type is sometimes also called **entailment**.

Subject/Actant1 Reciprocal (SAR)

The **Subject/Actant1 Reciprocal (SAR) Relation** relates two Actions. It is a type of Implication, but one which goes in the reverse direction: from actant 1 to the subject. For instance, if P Peter - A accompanied - P Susan, then P Susan - A was accompanied - P Peter.

Some SAR are **symmetrical**, i.e. they imply the reciprocation of the same Action from actant 1 to the subject (e.g., "was in the company (of sb)"). Some are **asymmetrical**, i.e. the reciprocated Action is different.

Actant semantics: Subject Semantics (SUS), Actant 1 Semantics (A1S), and Actant 2 Semantics (A2S)

Subject Semantics (SUS), **Actant 1 Semantics (A1S)**, and **Actant 2 Semantics (A2S)** are Relations each of which connects an Action to one or more Concepts **describing semantically the role of the holder of the actant slot**. E.g., A "to speak (to sb - about st/sb)" would have the SUS: C "speaker", A1S: C "participant in a conversation", and A2S: C "speech content".

These Relations describing actant semantics then allow to **study the roles of entities holding that actant slot**. E.g., we can inspect all women who have either the in-statement CLA C "preacher", or who are known to have A "preached", because A "preach" would have exactly the SUS: C "preacher". Therefore, the actant semantics enrich actant role holders with the respective roles implied by the Action.

These relations also allow to properly understand the **direction of action**: e.g., a text can say that `P Lucy - A listened - (to) P Mary`, or that `P Mary - A spoke - (to) P Lucy`. The relevant actant semantics will allow to identify, in both wordings, Mary as the speaker and Lucy as the listener. Thus, for research purposes, we can for instance safely produce a directed conversation tie going from Mary to Lucy rather than the other way round from any of the two wordings, and also harbour the **passive and active voice** correctly under the same roof.

Actant semantics are the CASTEMO way of ensuring **semantic role labelling**: it is done by properly defining any Action type's semantic valency, rather than losing time with labelling semantic roles in specific Statements.

Related (REL)

Related (REL) is the least specific Relation which allows to relate entities of any type by way of association.

Example: `C tea - REL - C afternoon`

For more specific semantic relations, a relevant specific Relation or Property should be preferred.

References

How best collect CASTEMO
data?

How best collect CASTEMO data?

Describe your data collection choices

Every data collection campaign, even the most comprehensive CASTEMO annotation, necessarily **makes choices**, and is **selective**.

In collaboration between several users, and also as time goes by, it becomes increasingly tricky to remember what data collection guidelines you used, what you included, what you skipped... while this is, obviously crucial for the **interpretation of results** based on the data collected. Therefore:

You should always describe your data collection choices.

Furthermore:

It is generally better to keep the description of data collection choices very close to the data themselves.

In CASTEMO knowledge graphs, we recommend to append the description of the collected data as part of the **metadata of the Territory** which holds this CASTEMO data. There is a **pre-defined section Protocol** under each Territory which allows you to capture the basic ones. Beyond that, you might want to describe your choices in a more narrative and comprehensive way in a **document** (webpage, Google Doc), which can be linked from the Protocol.

How best collect CASTEMO data?

"Same as above": Referencing information content in CASTEMO knowledge graphs

Referring to the content of another

Documents and statements often make references to other documents and statements to express that the content is the same, different, or related in other ways.

... same, different. Treat this here.

Referring to temporal and spatial information of another document

...

Epistemic level in the referencing property

General recommendations for epistemic level apply. Therefore:

- If the reference is **contained in the text**, the whole property should be at the textual epistemic level.
- If in this property, you are using an **analytical property type** (e.g. C "same content") to which something corresponds in the text, but the reference is not done with the same words, the involvement of this property type in the property statement will be at **interpretive level**.
- If the **property value** is a document (i.e. "the same thing as text X"), while the actual text is referring to what a previous *person* (not *document*) said, then its involvement in the property statement will be **interpretive**; if the document is being referenced and in virtually the same words as the label of the document has it, then the property value's involvement in the property statement will be **textual**.

Full-text annotation

InkVisitor supports full-text annotation in its Annotator component. This chapter will teach you:

- some basics of semantic annotation;
- use of Annotator in the InkVisitor application.

Import a full-text document and start the annotation work

Before starting to annotate, you need to import a full text in InkVisitor, create a Resource representing this full text, and link it to a Territory. This page describes the steps to follow.

1. Give a thought to your data management plan

Annotation creates an additional, XML-like markup layer over the full-text document. Therefore, you need to think where the InkVisitor-managed full-text belongs in your data management plan. You can always export the full-text with all of your annotation (anchors) from InkVisitor, so in that sense you are safe. However, since the moment you start annotating and making text corrections in InkVisitor, this InkVisitor-managed version should be updated with all enhancements.

Think about your full-text data management. Avoid creating two mutually outdated versions.

2. Clean the full-text document

You will usually want to import **the main text of a source**, without **editorial text** (we call it CAFE: Critical Apparatus, Footnotes, Editorial matter).

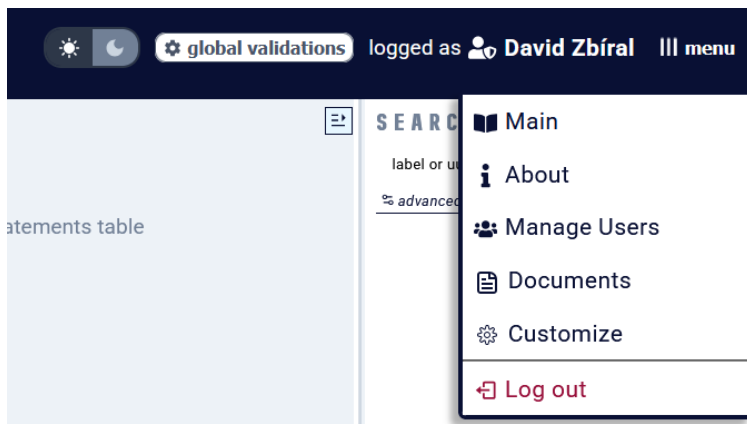
While you can also make edits in InkVisitor, it is generally advisable to import full-texts which are ready enough for work - with CAFE removed and the text (e.g. OCR) of sufficient quality.

InkVisitor accepts **plain text** and (since Summer 2025) **marked-up text** (e.g., XML).

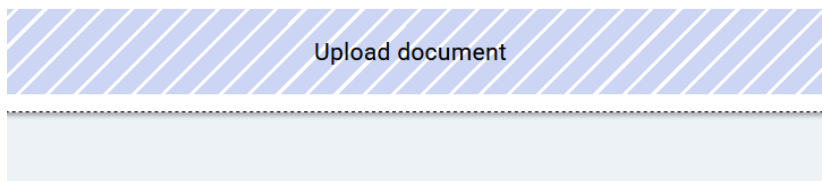
The characters "<" and ">" are **reserved markup delimiters** and must not be used for anything else in the full text. Remove those which are not markup delimiters or replace them with another character before full text import.

3. Import the full-text document

Open the document component from the main menu (top right corner).



Here, click on **Upload document**, locate the file on your drives, and confirm.



Your document will be uploaded.

4. Represent the full text with a Resource entity

The full-text needs to be represented by an **entity of the type Resource**. Of course, this entity is not the source (text) in general; it is **one specific textual version**, edited by a specific editor and processed with a specific workflow (OCR-ed, transcribed, or digital-born).

Therefore, **create an R entity representing the full-text document**. The entity can be named e.g. "Letter of Ebervin of Steinfeld to Bernard of Clairvaux (ed. Migne, 1879), MedHate-curated version".

After you create this Resource, **add a metadata description**.

Metaproperties

append

*

▼

another entity

C

represented Territory

T

Letter of Ebervin of Steinfeld...

+p

a

C

editor

P

Jacques-Paul Migne

+p

a

C

corrector

P

David Zbiral

+p

a

C

curator

P

David Zbiral

+p

a

C

digitization unit

*

▼

value

+

+p

a

C

text quality

*

▼

value

+

+p

a

Acknowledge the original editors and curators in the metaprops to give them proper credit in your data.

Save time and ensure data coherence by using a Resource **template** probably already created in your deploy for similar digitised resources.



5. Link your Territory to this Resource




This R (specific textual version) represents a Territory (text), which probably already exists (alternatively, you need to create it at this point). Now, you need to **link the Resource to the Territory** in two steps:



First, **open the Territory, switch to Annotator view, and find the correct R** representing the full-text document you want to link.



No territory selected yet. Pick one from the territory tree

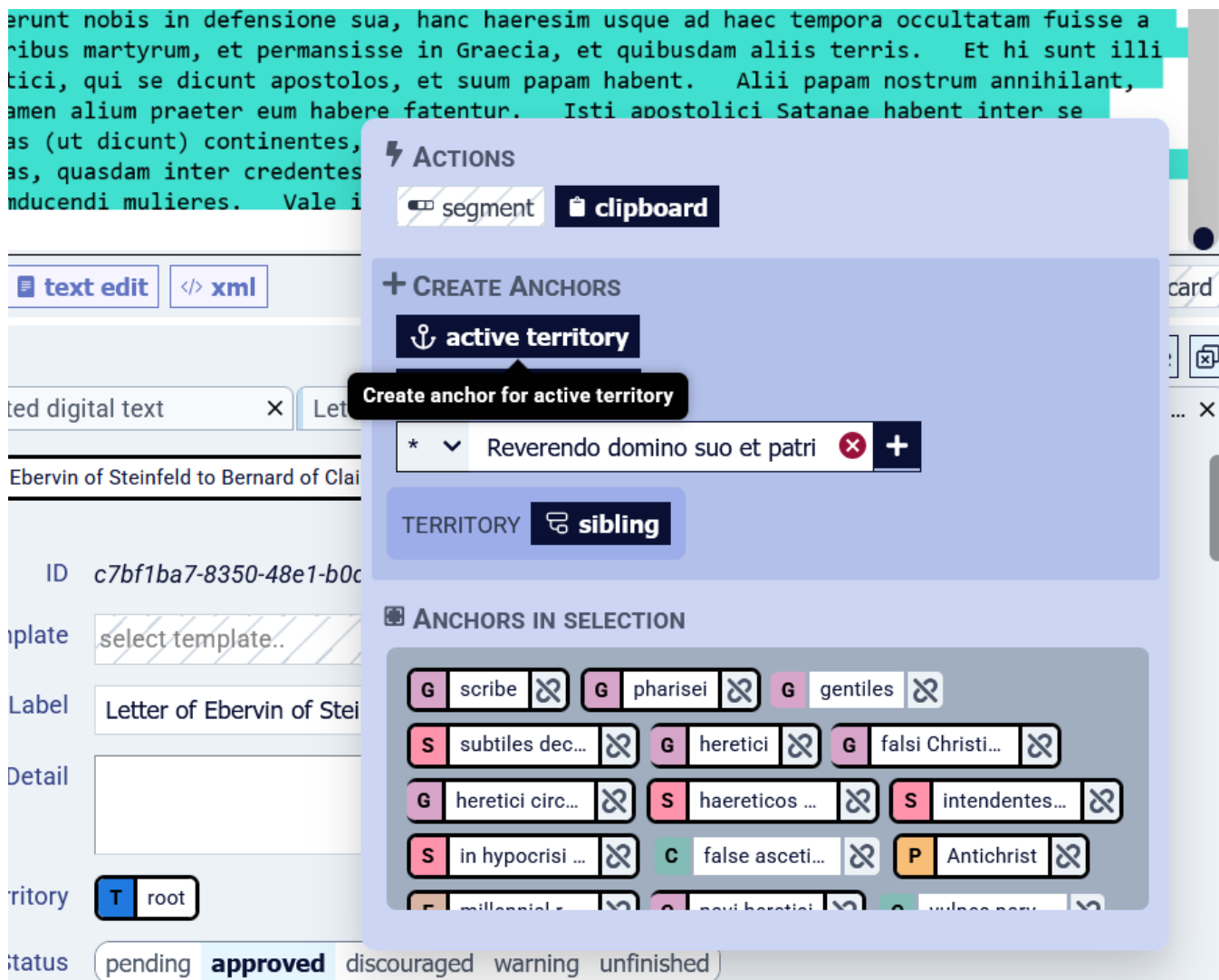
R eber|  

-  R Patrologia Lat...
-  R Iohannes Aqu...
-  R Letter of Eber...

 Letter of Ebervin of Steinfeld to Bernard of Clairvaux (ed. Migne, 1879), MedHate-curated version (English)
 MedHate-curated digital text text

Second - in order to confirm the relation and save it permanently - **add the anchor (start and end tag) of the Territory into the full-text itself:**

1. **Select the full text of this Territory** in the full-text document displayed, from beginning to end (using Ctrl+A, Shift+PgDn, or mouse).
2. A modal window appears. Anchor active territory in the full-text with the **button "active territory"** in the modal window.



6. Annotate

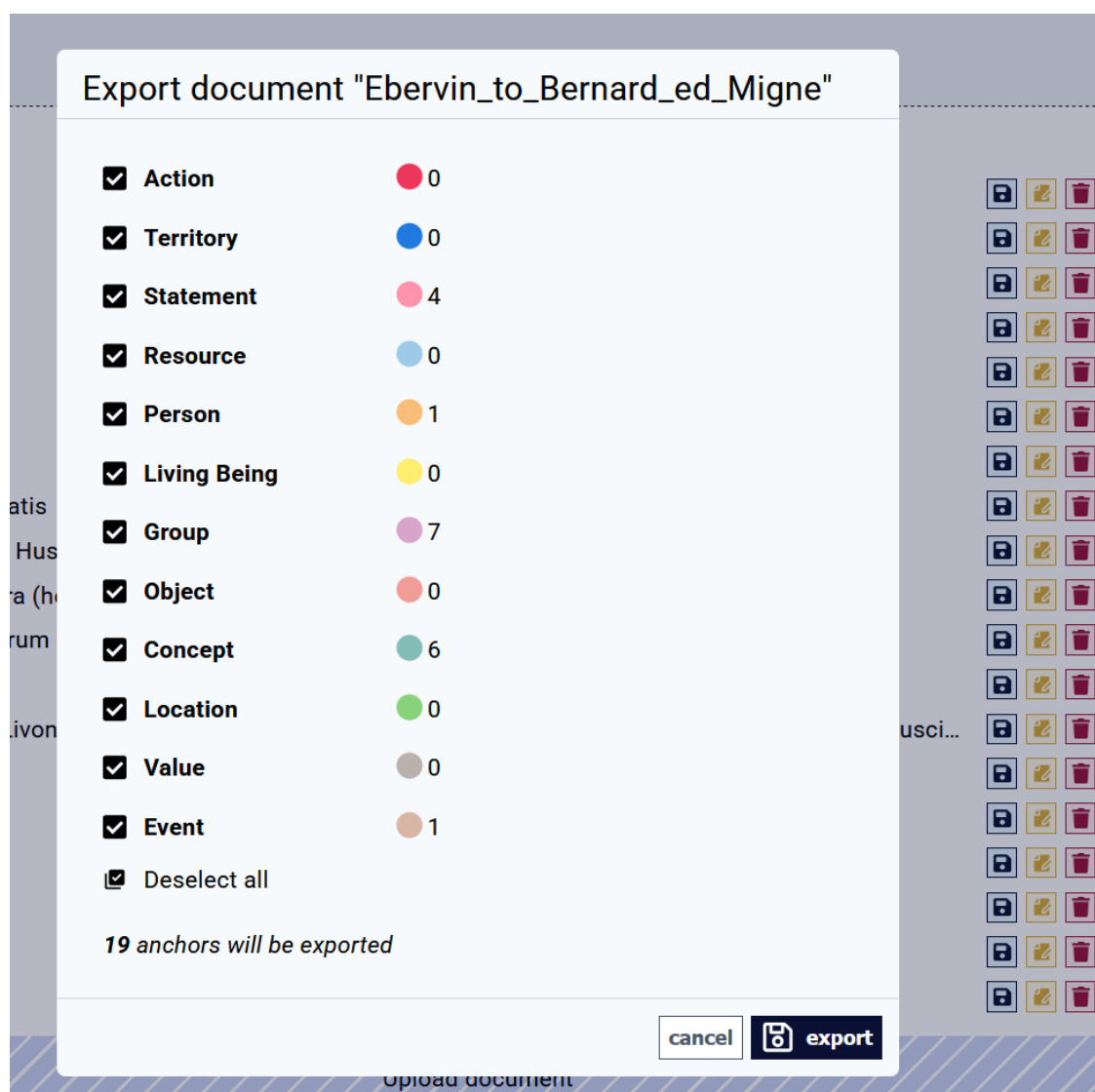
Now you are **ready to annotate**, that is, to add entity anchors in the full-text document. In the process, you will probably be creating new entities as you go, and using Relations to place them in proper semantic and ontological relations to then query them for research.

The anchors are the **start and end tag** of entities of any type, which are **stored in the full-text document** in an XML-like way using the entity's unique identifier (UUID). The following example shows the anchors visually in the InkVisitor Annotator interface, and in the XML-like code.

InkVisitor is a tool for annotating **digital texts, not images** (e.g., manuscript reproductions). If you need that, you will need to use a different annotation software.

7. Export

For **backup and analysis**, you will need to export your full-text document alongside the anchors you created in the text. This is done in **Documents** (see main menu in the top right) through the floppy disk icon. Typically, be sure to include **all anchors** in the export.



If you export the full text without including anchors of all entity types and then overwrite the InkVisitor-managed text by a reimported full text (e.g. after making some batch text corrections outside of InkVisitor), the anchors you haven't exported will obviously be lost forever.

Querying CASTEMO knowledge graphs

Now, time to get knowledge out of the knowledge graphs. This chapter categorizes some useful queries to validate your data, and start digging information from the CASTEMO knowledge graphs stored in RethinkDB.

Finding inconsistent and invalid data

This is the collapsible text.

For various reasons, such as data import or bugs of some version of the interface, a CASTEMO knowledge graph can contain **inconsistent data**. It is thus important to identify this data and correct the inconsistencies either manually or from a script.

Get all entities involved in more than one synonym cloud

In the CASTEMO data model, one entity can be only involved in one synonym cloud.

ReQL

```
r.db("inkvisitor")
  .table("relations")
  .filter({ type: "SYN" })
  .getField("entityIds")
  .reduce(function (acc, val) {
    return acc.setUnion(val);
  })
  .default([])
  .map(function (ide) {
    return {
      id: ide,
      count: r
        .db("inkvisitor")
        .table("relations")
        .filter({ type: "SYN" })
        .getField("entityIds")
        .filter(function (syn) {
```

```
        return syn.contains(ide);
    })
    .count(),
    };
})
.filter(function (a) {
    return a("count").gt(1);
});
```

R

Empty language: get labels of all concepts with empty language

ReQL

```
r.db("inkvisitor")
  .table("entities")
  .filter({ language: "", class: "C" })
  .getField("label");
```

Querying CASTEMO knowledge graphs

Querying CASTEMO knowledge graphs in Neo4j

Querying CASTEMO knowledge graphs

Querying with relations

Data import

Parsing instructions for the import of tabular data

This doc describes the parsing/import instructions used in the google sheet tables meant to be imported to inkVisitor DDB1, which conforms to DDM (=DISSINET DATA MODEL).

The instructions were mostly improvised in the conversations between DZ and TH, they changed a lot during the development of the inkVisitor data model and parsing. E.g. the system and terminology probably are far from ideal. Sorry for that. :)

Prerequisites

It is helpful to understand the issues around [properties, metaproperties and 2nd order metaproperties](#).

Basics

The first **four rows** of the google sheet table (designated for parsing+import to inkVisitor DDB1) form the **parsing header**.

The **fifth** row contains a header row with column **labels**.

Each other following row represents one **main entity**, e.g. "Person" or "Location".

The foundational part of import instructions parsing instruction **keywords** (e.g. *discard*, *inside*, ...), sitting in the 4th row.

The four rows are:

1. **comment**- anything can be here, it is for humans, mostly used for the definition of the "special" instruction
2. **target object** - for parameters of relational instructions, (e.g. "id"), signals target to which this column is supposed to hooked
3. **type** - for parameters of relational instructions (e.g. "R0002"), for the first part of the property type-value pairs, i.e. dog has "color" (**type**) "brown" (value)
4. **parsing keyword** (e.g. "reference_part") - primary information on what to do with the column data

5. the label /name of the column

	A	B	C	D	E	F	G	H	I	J	K
1					discard - unification						
2									id	id	
3									R0002	Classificati	
4	inside	inside	inside	inside	discard	discard	discard	discard	reference_p	relation	discard
5	id	label	label_lang uage	entity_logical_ty pe	resource_ id	document no	page_range start	page_rang e_end	page_range concatena	class_id	class_label
6	E0137	[Rubrica]	Latin	definite	R0002		52	52	52	NA	NA
7	E0138	[Preambula]	Latin	definite	R0002		52	52	52	NA	NA

Generally, there are three types of instructions

- simple, they do not need other parameters (e.g. *inside*, *discard*)
- relational, they specify relational data and **need** other parameters (e.g. *propvalue*, *relation*)
- special, they are specified in natural language in the comment row

Here are all possible keywords:

simple	inside, discard, hooked-inside
relational	relation, reference_part, propvalue, proptype, proptype_2nd, propvalue_2nd, hooked-relation, hooked-propvalue
special	special

All columns in the table meant for parsing and import should have an instruction header!

Meaning of the parsing instruction keywords

Simple

inside	the content in the column is meant to be directly inside the entity objects, e.g. for columns like "label" or "note", that target entity need to have such attribute specified by the data model
discard	the content in the column is just ignored
hooked-inside	the content in the column is meant to be inside of some " embedded object ", which is always defined by the preceding <i>special</i> column

Relational

	description	param1 type	param2 target object	the cell can contain
relation	for making DDM "Relation"	name of the relational type (e.g. "Classification", "Identification")	can be <i>empty</i> but usually contains " id " as a signal that this entity is connected to the main entity	a concept legacyId, e.g. C3166 (defendant deposition)
reference_part	for making DDM references to resources	resource ID (e.g. R0002)	can be <i>empty</i> but usually contains " id " as a signal that this entity is connected to the main entity	string of the pages, e.g. "v216", which is transformed into Value object
propvalue	for making metaproperty , where the type is A. fixed for whole column OR B. defined by main entity field with <i>proptype</i> instruction	a concept, which defines the type, e.g. C0316 (occupation)	empty OR name of the column with <i>proptype</i>	entity legacyId, or string which is transformed into Value object
proptype	for making metaproperty , where the type can differ		empty or can contain "id"	the C entity
proptype_2nd	for making 2nd metaproperty , where the type can differ		name of the <i>propvalue</i> column, to which this 2nd order property is hooked	
propvalue_2nd	for making 2nd metaproperty , where the type is A. fixed for whole column OR B. defined by main entity field with <i>proptype</i> instruction	a concept, which defines type, e.g. C0316 (occupation)	name of the <i>propvalue</i> column, to which this 2nd order property is hooked	entity legacyId, or string which is transformed to Value object
hooked-relation	as <i>relation</i> , but it is controlled from special instruction	as relation	can be empty, but usually contains the name of the column, which has <i>special</i> instructions	
hooked-propvalue	as <i>propvalue</i> , but it is controlled from special instruction	as <i>propvalue</i>	can be empty, but usually contains the name of the column, which has <i>special</i> instructions	

Special

special	Conforms to the particular parsing function, which is fully custom and based on the instructions.
---------	---

InkVisitor installation on the server

This chapter is intended for your IT support. It describes how to deploy the InkVisitor application on a server so that you can start using it.

How to deploy your own instance of InkVisitor

Deploy with Docker

You can use docker to deploy the InkVisitor application

1. Install [docker](#).
2. Install [docker-compose tool](#).
3. Clone | Fork | Download the Inkvisitor [repository](#).
4. Prepare `.env` files for servers listed under `env_file` sections. Check the server's [README.md](#) and [example.env](#) files for more information.
5. To prepare the necessary configuration files for the client application, you should identify the appropriate environment variables (ENV) under the `build -> args` section and then use them to create the `.env` files. You can see the server's [README.md](#) and [example.env](#) files to ensure you have included all the necessary configuration information.
6. Run the database - either as a service or containerized using `docker-compose up -d database`
7. Build app image (will also be done in next step if not available) `docker-compose build inkvisitor` (or `inkvisitor-<env>`).
8. Run the containerized application with the command `docker-compose up inkvisitor` (or `inkvisitor-<env>`).

Deploy by packages

The InkVisitor codebase consists of three interconnected packages (parts) - the client application, the server, and the database. You can deploy those packages individually if you do not want to use Docker. In each step, make sure to have the appropriate `.env.<env>` file accessible - see the Readme.md file in the package for more information.

1. Client application

The client application runs on static files - html/css/js + additional assets. These files need to be moved to your HTTP server by:

1. Build the frontend app by `npm run build-<env>` to create/update the `dist` folder
2. Copy contents of `dist` folder to the directory used by your HTTP server.

2. Server

The server is also built in Javascript, using mainly the Node + Express libraries. You need to first build the application, move the build to your server and run it from there.

1. Run `yarn run build` to transpile the code.
2. Move the `dist` folder to your server that supports the Node.js environment.
3. Do `ENV_FILE=<env> yarn run start` to run the built application with a loaded `.env.<env>` file.

3. Database

Follow tutorials on the [official page](#) to install RethinkDB on your machine. Then, use the import script to create the database structure and (optional) import some testing data by running ``npm run import`` and following the information in the prompt.

Firewall

Make sure the ports required by each application are not blocked. Required ports are listed in [docker-compose.yml](#). Examples:

1. [ufw](#): `ufw allow <port>`
2. [firewalld](#): `firewall-cmd --zone=public --permanent --add-port=<port>/tcp`

Setup for additional system specific features (reverse proxies etc) are beyond the scope of this readme.