ALMA MATER STUDIORUM BOLOGNA - University of Bologna
Knowledge Engineering project

Translation Coherence

Authors:
Lorenzo Mario Amorosa - lorenzomario.amorosa@studio.unibo.it
Michele Iannello - michele.iannello@studio.unibo.it
Andrea Lavista - andrea.lavista@studio.unibo.it

1. Overview and preliminary steps

This project aims at exploiting **Knowledge Engineering** technologies to **compare translations** across multiple languages by formal reasoning on them. Our goal is to understand which subtle variations occur when going back and forth from a language to another using off-the-shelf translators (such as DeepL).

We have chosen the open-source **Europarl** parallel corpus for this task. In particular we have taken the first 6 sentences from the English dataset and we have subsequently translated them to German, Italian and Chinese, and then we have translated them back to English. The choice of the languages is not incidental: we aimed at observing whether more differences pop up by translating towards more and more exotic languages with respect to English, thus we picked up a West Germanic language - German - a Latin language - Italian - and a non-European language - Chinese.

Once we have translated the sentences as described above, we employed the **machine reader** FRED to encode them as knowledge graphs. Starting with 6 sentences translated from 4 languages we ended up with 24 different ontologies.

The attempts we made to address the problem of computing the **semantic differences** between a pair of ontologies are described in the following sections.

All the code is available on GitHub.

2. Ontology alignment with LIMES

We firstly tried to use LIMES, which is an already well known existing tool to **align ontologies**, but we immediately felt uncomfortable with it, since we found it not adequate and suited for our purposes. The main reason is that what we had to do was *not exactly ontologies alignment*, but rather a sort of *semantic differences classification* between two given knowledge graphs. In other words, we managed to obtain a resulting ontology from an input pair of ontologies using LIMES, but it was not effective to reach our goals. In fact, we only retrieved all the nodes present in both the ontologies using LIMES, exploiting criteria on their labels.

As a consequence, we moved towards a more complex approach employing techniques such as **Graph Matching**, where we basically programmatically traverse the graphs and recognise semantic differences and similarities in ontologies searching for **recurrent semantic patterns**. In other words, we have defined our own reasoner which generates an ontology by analysing the ontologies of a sentence translated from different languages. This approach will be discussed in the following section.

3. Custom approach towards semantic analysis

We developed a programmatic approach to perform **semantic analysis** on the ontologies. We relied on the Python library `RDFLib` to import the graphs produced by the machine reader and to process its content.

The output of the semantic analysis is an ontology, called *result graph* in what follows, consisting of a set of triples which describe the semantic relations among the two compared graphs (e.g. synonymy relations). The result graph can bind already defined nodes, present in the input graphs, or new nodes defined ad-hoc for our purposes and described in the next sections.

In line with the principles of **eXtreme Design methodology**, we used *competency questions* to better identify suitable *Ontology Design Patterns* (ODPs) in order to design our result graphs. These are our competency questions:

1. What are the differences between the two ontologies?
2. Which pairs of nodes are related with synonymy relation?
3. What are the differences between the T-Box of the two ontologies?
4. Which are the synonymy relations in the A-Box of the two ontologies?
5. Which expressions are different in the two ontologies?

After the creation of the result graphs we tested them with the **SPARQL** queries relative to our competency questions, which are:

1)

```
SELECT ?s ?p ?o
WHERE{
  {?s tc:different ?o} UNION
  {?s tc:synonymy ?o} UNION
  {?s tc:differentHierarchy ?o} UNION
  {?s tc:similarHierarchy ?o} UNION
  {?s tc:differentExpression ?o}
  ?s ?p ?o .
}
```

2)

```
SELECT *
WHERE{
  ?s tc:synonymy ?o .
}
```

3)

```
SELECT ?s ?p ?o
WHERE{
  {?s tc:different ?o .
  ?s rdf:type tc:ClassConcept} UNION
  {?s tc:synonymy ?o .
  ?s rdf:type tc:ClassConcept} UNION
  {?s tc:differentHierarchy ?o} UNION
  {?s tc:similarHierarchy ?o}
  ?s ?p ?o .
}
```

4)

```
SELECT ?s ?o
WHERE{
  ?s tc:synonymy ?o ;
  rdf:type ?type
  FILTER(?type != tc:ClassConcept)
}
```

5)

```
SELECT *
WHERE{
  ?s tc:differentExpression ?o .
}
```

where the prefixes we used are:

```
PREFIX
tc:<https://w3id.org/stlab/ke/amiala/translation_coherence/>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

Then, in order to make the statements in our results graphs we had to define a vocabulary with classes and predicates suitable for our purpose, which is to represent knowledge concerning translation coherence. We modeled our vocabulary using Protégé (an open-source ontology editor): it is publicly available in `.owl` format at this link.

3.1 Building the graphs

A pair of graphs produced by the machine reader are taken for the comparison. These graphs are firstly cleaned from the unnecessary information, such as the data regarding the textual reference offset, and then enriched with the `rdfs:label` predicate to bind each node with the corresponding textual fragment. We developed a custom plot function using NetworkX, but we ended up using the service available here to plot the graphs due to the greater ability of the latter service to portray complex relationships.

Up to now, each node carries no more information than its IRI and label: not much when it comes to semantic tasks such as comparing two synonyms or two different inflected forms of the same word. In order to accomplish a pairwise *semantic* comparison between two graph nodes whatsoever, it was necessary to firstly perform the **lemmatization** of the words occurring in the sentences, which we did thanks to Python Library spaCy: indeed, we created a Python dictionary to store the **lemma** of each node's label in order to be able to easily retrieve them through a look-up operation.

3.2 Comparing the graphs

The process of graphs comparison is quite complex. The **core strategy** is to search for subgraphs that are present in both graphs to identify nodes which are either shared (and thus semantically **equal**) or semantically **different** but accomplish the *same role* in the sentences. In other words, some *stronger* equivalences are taken as certain and used as starting points to explore the graphs side by side (Section 3.2.1), searching for both equivalences and differences. The systematization of this comparison is achieved through the identification of recurrent subgraph structures, referred to as *patterns* (Section 3.2.2-3).

The overall structure of the comparison ultimately consists of two nested **loops** that have the following structure (pseudo-code):

```
frontiers = find_starting_points(graph1, graph2)
while(frontiers):
     while(frontiers):
          propagate_equivalences(graph1, graph2, frontiers)
          apply_safe_patterns(graph1, graph2, frontiers)
     apply_unsafe_patterns(graph1, graph2, frontiers)
```

The first step is to find the *strongly equivalent* nodes (and their neighbours) and the *equivalent classes* (Section 3.2.1) and store them in a list of pairs called *frontiers*. ***Patterns*** are subsequently applied on *frontiers* nodes in order to enrich the information about the graphs comparison. In particular, after the propagation of the equivalences (Section 3.2.2.1) a set of *safer patterns* such as *negative verbs* (Section 3.2.2.3) are applied more frequently in an **inner loop**. When the safest *patterns* do not produce any additional output, some *more error prone patterns* such as *find synonymy classes* (Section 3.2.3.2) are applied to derive more information.

In these loops a set of *patterns* is applied on the nodes already marked as *equivalent* or *strongly equivalent*. When a match occurs, each *pattern* produces new triples for the result graph and possibly updates the *frontiers* with the new nodes involved in the *pattern*, so that *patterns* can be applied on new nodes in the following iteration.

3.2.1 Strongly equivalent nodes and equivalent classes

The very first step of the algorithm implemented in function `compare_graphs` is finding the *strongly equivalent* nodes (`:stronglyEquivalent` predicate), which basically are those nodes which **share at least 3 identical triples** in both graphs. We chose these nodes as *anchors* for the propagation of equivalences and the application of *patterns*, due to the reliability of their match. In our tests - i.e. using **short phrases** with few subordinates - these nodes resulted to always have the same semantic role in both the test sentences, meaning that the system *translator* + *machine reader* lead for both translations to the same representation of words within sentences. As a consequence, we can state that the *strongly equivalent* nodes define portions of text **shared** across sentences.

For example, the node `fred:disaster_1` in *Figs. 1-2* is identified as a *strongly equivalent* node, since it is involved by 3 triples shared among the 2 graphs (the arcs representing the predicates of those triples are marked in red).

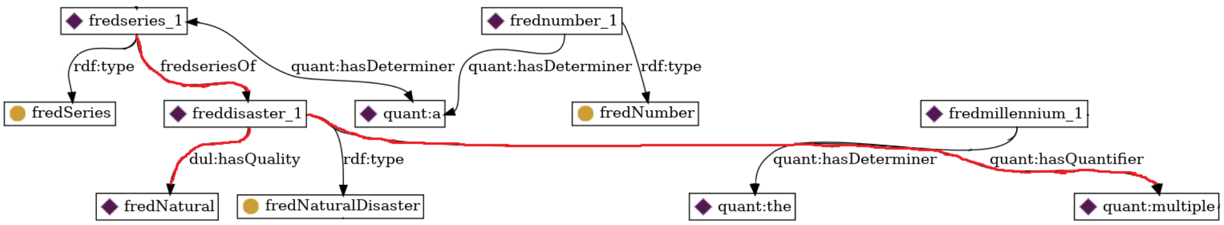

Fig.1. en_sentence2 subgraph

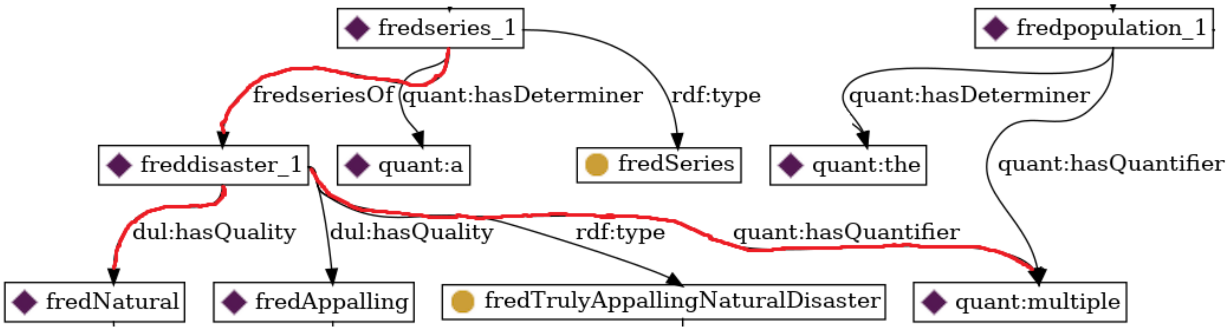

Fig.2. en_it_en_sentence2 subgraph

Afterwards, all the classes which have the same name in the 2 graphs are marked using the predicate `:equivalent`. This last predicate represents that it is very likely that the subject and the object (which are in different graphs) have the same semantic role in the sentences.

3.2.2 Patterns in the inner loop

3.2.2.1 Find equivalence relations

The first pattern *find equivalence relations* consists in finding all the pairs of nodes which respect these conditions:

1. the *lemmas* of their label are the same;
2. they are respectively *linked* to nodes constituting a pair in the list *frontiers* (i.e. already classified as equivalent) with the *same predicate* in both the graphs.

In this way the network of equivalent subgraphs is extended including more related nodes.

For instance, consider Figs.3-4 and assume that the nodes `fred:disaster_1` - present in both graphs - were already marked as *strongly equivalent*: the node `fred:Natural` in the first graph is accordingly marked as *equivalent* to node `fred:Natural` in the second graph, since it is linked in both graphs to `fred:disaster_1` with the predicate `dul:hasQuality` (marked in red).

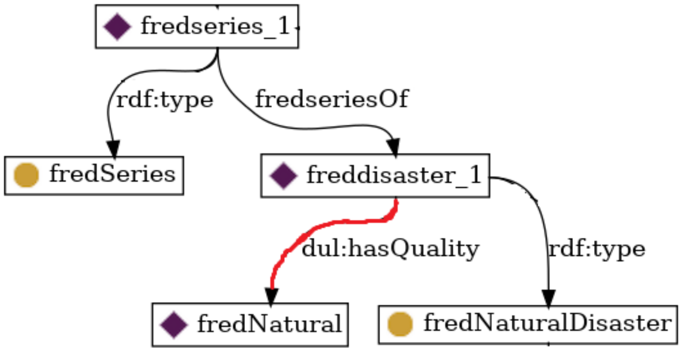

Fig.3. en_sentence2 subgraph

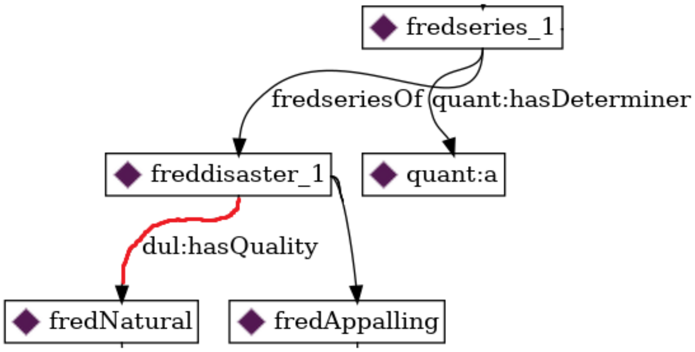

Fig.4. en_it_en_sentence2 subgraph

3.2.2.2 Find synonymy relations

The next pattern is called *find synonymy relations* and its purpose is the same as the previous one, except for the first condition: the nodes in the graphs are not required to share the same lemma anymore, instead it is enough that their lemmas are **synonyms**. The network of classified subgraphs is extended including more related nodes using the predicate `:synonymy`.

The criteria for the synonymy are based on WordNet® - deployed thanks to Python library NLTK: we investigated different alternatives, trying different metrics provided by WordNet for word similarity. In our implementation, two words are considered synonyms if one or more of the following condition holds:

1. they share at least one synset (synsets are "*sets of cognitive synonyms, each expressing a distinct concept*")
2. their wup (*Wu-Palmer*) similarity is at least 0.85
3. their path similarity is at least 0.45

Further information about these metrics are described in the NLTK documentation. We also implemented the synonymy criteria using GloVe, but it was not the optimal solution (for instance, not only *synonyms*, but also *opposite* words have similar embeddings).

For instance, consider Figs 5-6 and assume that the nodes `fred:suffer_1` - present in both graphs - were already marked as *strongly equivalent*: the nodes `fred:people_1` and `fred:population_1` are accordingly marked as *synonymy* since they are linked in both graphs to `fred:suffer_1` with the predicate `vn.role:Experiencer` and their lemmas match the synonymy criteria.

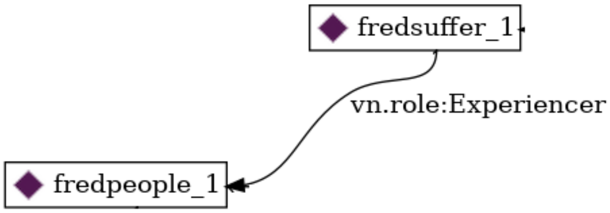

Fig.5. en_sentence2 subgraph

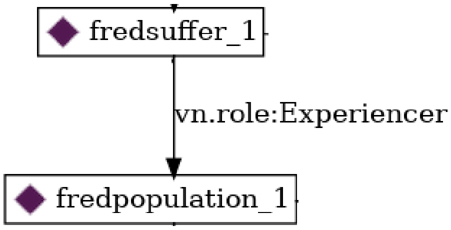

Fig.6. en_it_en_sentence2 subgraph

3.2.2.3 Negative verbs pattern

In the translation process some expressions happen to be mutated, for instance the predicate "didn't materialise" can be rendered with the expression "failed to materialise" or also "failed to appear". The pattern *negative verbs* aims to capture these last nuances. In particular, the **N-ary relation Logical ODP** is deployed for the representation of the *negative verbs pattern*. Possible encodings of the previously mentioned expressions are shown in Figs. 7-8: the expression “failed to materialise” is encoded as the triple (`fred:fail_1, vn.role:Theme, fred:materialise_1`), while “didn’t materialise” as (`fred:materialise_1, boxing:hasTruthValue, boxing:False`).

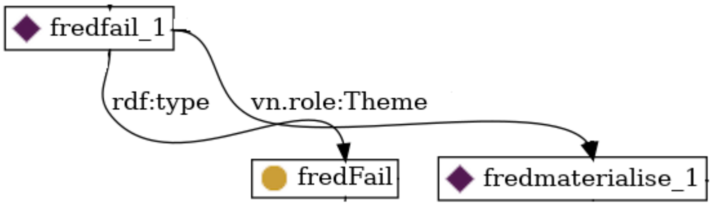

Fig.7. en_sentence2 subgraph

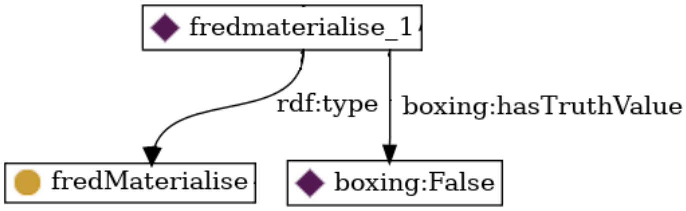

Fig.8. en_it_en_sentence2 subgraph

The pattern involves the creation of two individuals in the result graph, `:expression_1` and `:expression_2`, which implement the reification of the N-ary relations. Each `:expression_i` has relations with all the nodes involved with *ad-hoc* predicates, as it can be seen by inspecting the example in Fig. 9 (in the first graph with `fred:fail_1` and `fred:materialise_1` and in the second one with `fred:materialise_1` and `boxing:False`). Finally, the two `:expression_i` are linked with the predicate `translation_coherence:differentExpression`.

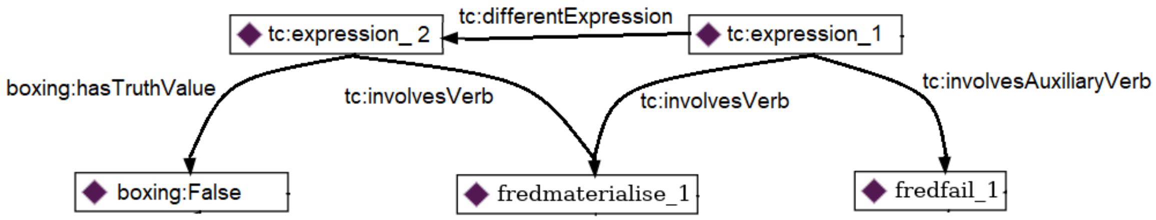

Fig.9. en_VS_en_it_en__sentence2, negative verbs pattern

3.2.2.4 Class-subclass equivalence pattern

We designed this *pattern* to address a variety of scenarios involving a pair of nodes in *frontiers* (which we refer to as `(n1,n2)` for the sake of simplicity), including:

- **n1** and **n2** are involved in a structure of the kind presented in Fig. 10, i.e. a certain resource appears as **individual** in one graph and as **class** in the other one[1]: in this case, the two corresponding individuals are linked through the `:equivalent` predicate.

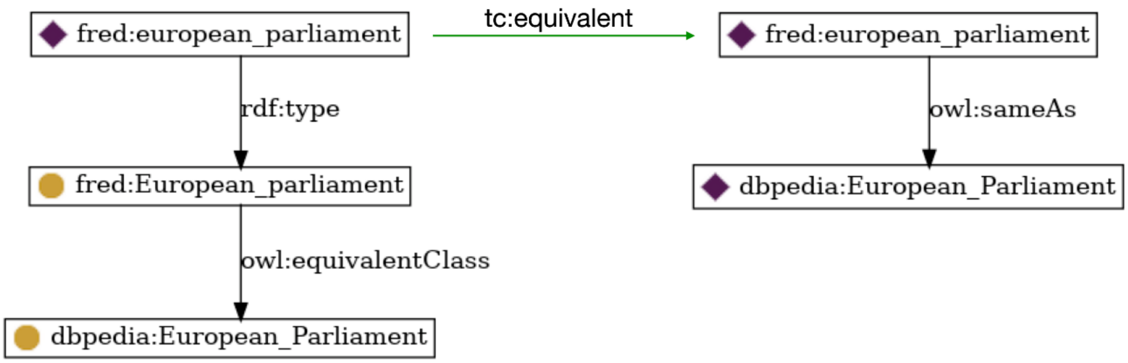

Fig.10: en_sentence1, en_it_en_sentence1

- **n1** and **n2** are members of a hierarchy of classes (or instances of a class that is) that are found either **pairwise equivalent** or **synonyms** (taking only *lemmas* into account): in this case, the corresponding predicate is used to respectively link the classes (see Fig.11).

[1] We observed this behaviour only on resources defined in the namespace DBpedia.

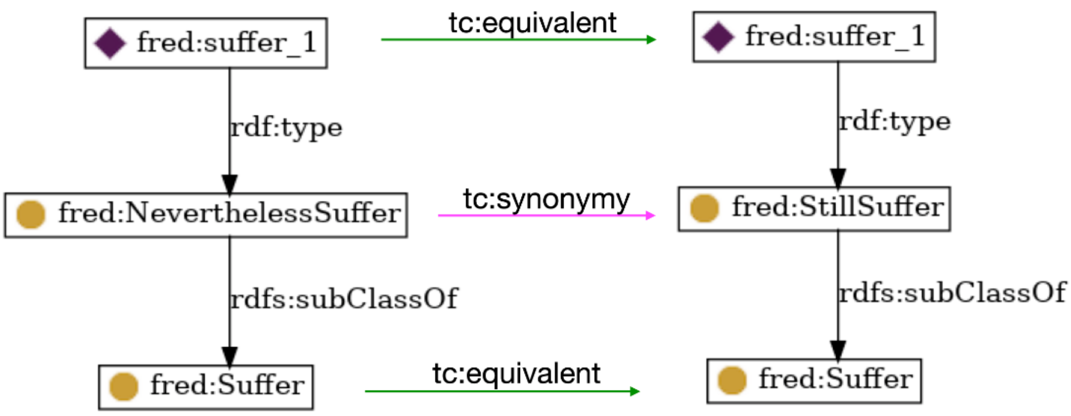

Fig.11: en_de_en_sentence2, en_cn_en_sentence2

- **n1** and **n2** are either **different** (Section 3.2.3.1) or members of two **non-straightforwardly relatable** hierarchies of classes. In this case two individuals `:hierarchy_1` and `:hierarchy_2` are created in the result graph to implement an **N-ary relation**, much like in Section 3.2.2.3: the *ad-hoc* predicate used to link them to the involved classes is `translation_coherence:hasHierarchyMember` (along with its inverse `translation_coherence:isHierarchyMemberOf`), while the two hierarchies are related either with `translation_coherence:similarHierarchy` or with `translation_coherence:differentHierarchy`. See examples on Figs.11-12.

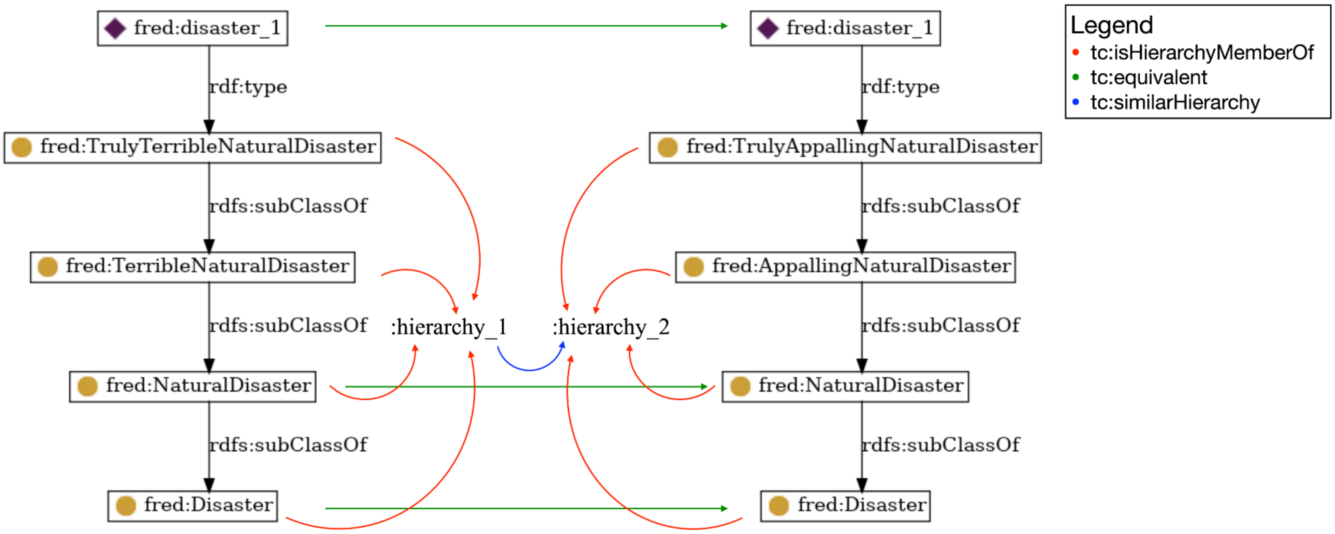

Fig.11: en_cn_en_sentence2, en_it_en_sentence2

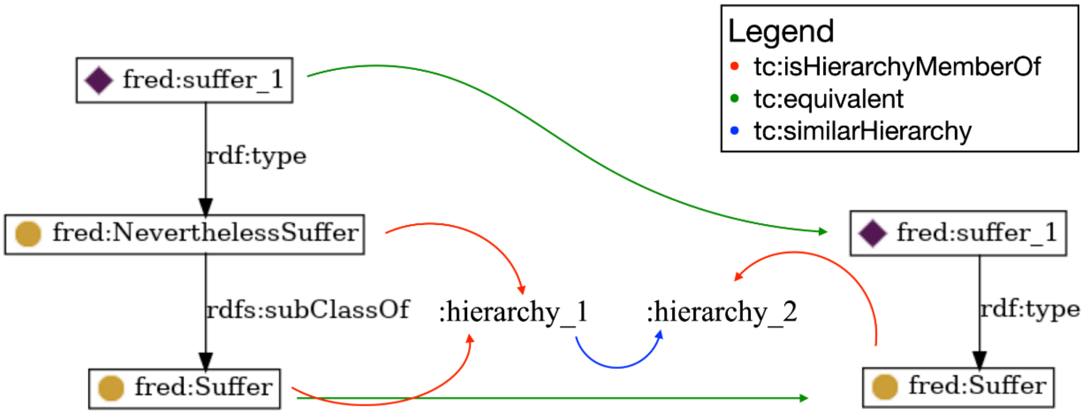

Fig.12: en_de_en_sentence2, en_sentence2

3.2.2.5 Check multiples pattern

This *pattern* captures two **differently formulated** expressions, much like *negative verbs* (Section 3.2.2.3), this time involving quantifiers: we noticed that words such as *series*, *number*, *group*, etc. can lead to different representations.
As an example we can observe Fig. 13: the piece of text "*a series of natural disasters*" is found equal in both translations, but in the output of the machine reader `fred:disaster_1` is related in one case with `quant:multiple`, while in the other one with `fred:series_1`. Just like before, two individuals `:expression_1` and `:expression_2` are in charge of implementing the reification of the N-ary relations, while `translation_coherence:involvesNoun` and `translation_coherence:involvesMultiple` are used as *ad-hoc* predicates. Moreover, the expression involving the quantifier is linked to it through the predicate `quant:hasQuantifier`, and the two expressions are linked through the usual `translation_coherence:differentExpression`.

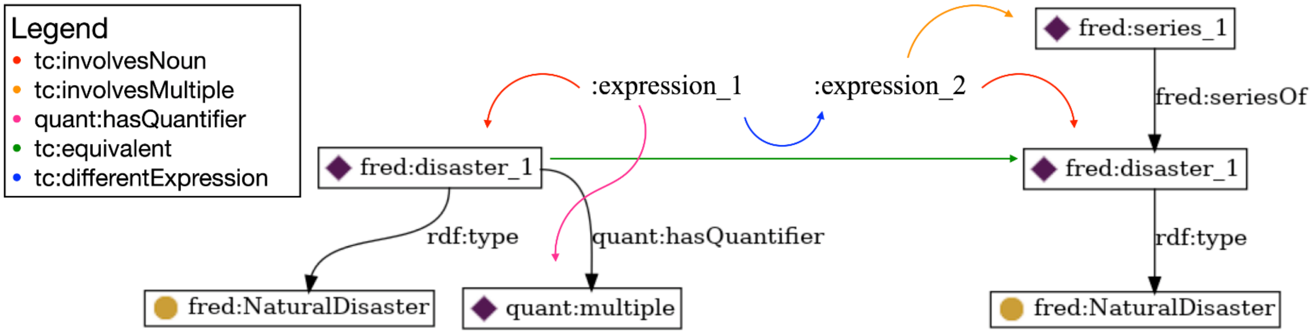

Fig.13: en_de_en_sentence2, en_sentence2

3.2.3 Patterns in the outer loop

When the patterns present in the inner loop do not provide any additional match, some ***less safe*** *patterns* are applied, so that new matches are produced and the pattern matching process is triggered again. The reason why the following patterns are defined as *less safe* is that they are implicitly less constrained and sometimes produce more matches than expected, thus we did not blindly trust them.

3.2.3.1 Find binary difference relations pattern

This pattern is called find *binary difference relations* and its purpose is to classify nodes in the two graphs that are not *equivalent* (their lemmas are not equal), neither *synonyms* (their lemmas are not synonyms), even if they seem to accomplish the same role in the ontologies.

In particular two nodes are classified with the predicate `:different` if the two nodes have *different* and *not synonym* lemmas, but are connected respectively, with the same predicates, to two nodes that are already marked as *equivalent* (or satisfy the equivalence condition).

For instance, considering Figs 14-15 and assuming that the nodes `fred:session_1` - present in both graphs - are already marked as *equivalent*: then the nodes `fred:adjourn_1` and `fred:interrupt_1` are marked as *different* since they are linked in both graphs to `fred:session_1` with the same predicate (`boxer:patient`) and they are linked in both graphs to `1999-12-17` (the same literal) with the same predicate (`fred:on`).

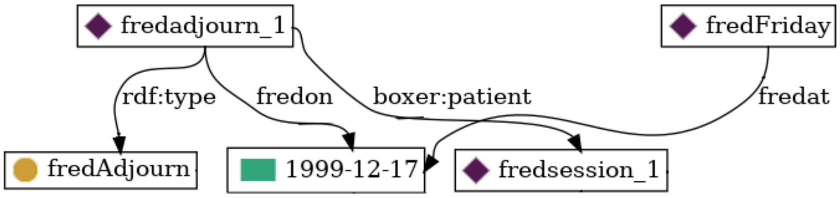

Fig.14: en_sentence1

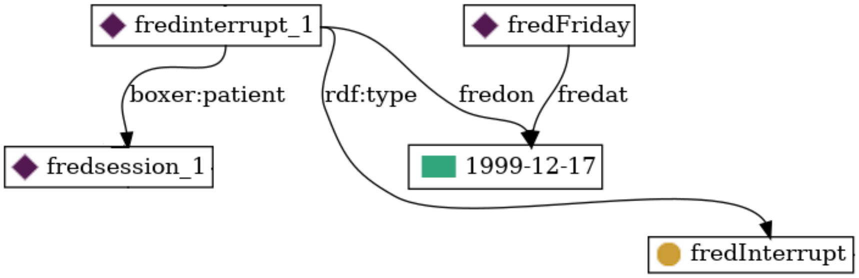

Fig.15: en_it_en_sentence1

3.2.3.2 Find synonymy classes pattern

Whereas the classes which share the same lemma are already marked as *strongly equivalent*, the classes whose lemmas are synonyms are not in any relation until this point. In fact, this pattern enlarges the list of *equivalent* nodes by adding all those classes whose lemmas are synonyms, in order to have more starting points from which to propagate the other patterns.

3.2.4 Patterns outside the loops

Once that either the patterns in the inner loop and in the other loop do not produce more matches, it is likely that some nodes are left out from the computed semantic relations. Only on these nodes, two final patterns are applied.

3.2.4.1 All different relations pattern

This pattern adds to the result graph a triple for each node which belongs to *both* the input graphs and *does not share any triple* in which it is involved between the two graphs. This happens often when words are arranged in a different order within the two sentences.

For instance, in the sub-sentence in `en_it_en_sentence2` "*a series of truly appalling natural disasters*" and the sub-sentence in `en_sentence2` "*a series of natural disasters that truly were dreadful*", encoded as shown in Figs.16-17, the node `fred:Truly` appears with different surrounding words and none of its relations is shared across the two input graphs. Consequently, the nodes `fred:Truly` present in both graphs are linked through the predicate `translation_coherence:differentContext`.

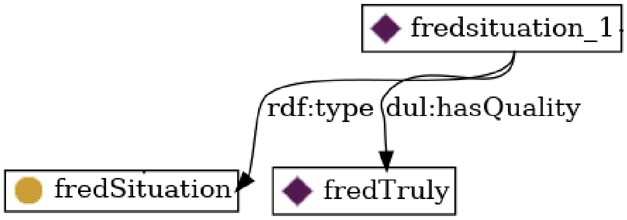

Fig.16. fred:Truly in en_sentence2

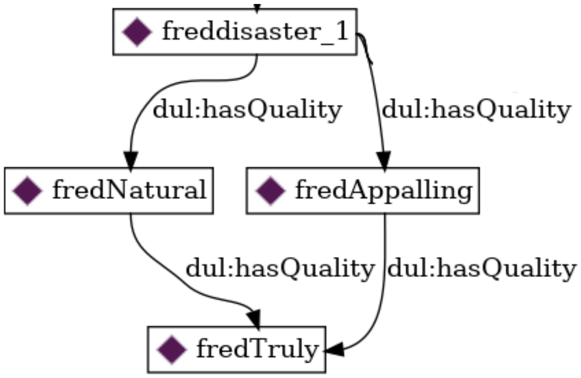

Fig.17. fred:Truly in en_it_en_sentence2

3.2.4.2 Only in one graph pattern

This pattern adds to the result graph a triple for each node which belongs to *only one* of the two input graphs and that is not already included in any triple of the result graph. When it comes to the comparison between `en_sentence2` and `en_it_en_sentence2`, for instance, the node `:Nevertheless` is found only in the second graph and it is not involved in any triple in the result graph, so the triple `(:Nevertheless, translation_coherence:onlyIn, :en_it_en_sentence2)` is added to the result graph.

3.3 Apply intensional reification

In our result graphs we often need to make statements about classes (to state that two classes are *equivalent*, *synonym*, etc.); we thus implemented the **intensional reification ODP**, in order to have individuals that allow us to talk about classes. In Fig 18 we show how we implemented this pattern.

The rectangles represent individuals, while rounded rectangles classes. In this example we show the reification of the two classes present in the left part of the figure: *GenericClass* and *GenericSubclass*, connected by the *subClassOf* predicate.

Our process of reifications will lead to the creation of individuals of type *ClassConcept*. Then we connect each individual to the class it represents through the predicate `rdfs:seeAlso` and we keep the information about subclass relations, using the predicate `translation_coherence:parentClassConcept`. To conclude we declare that the class *ClassConcept* is disjoint with all the reified classes.

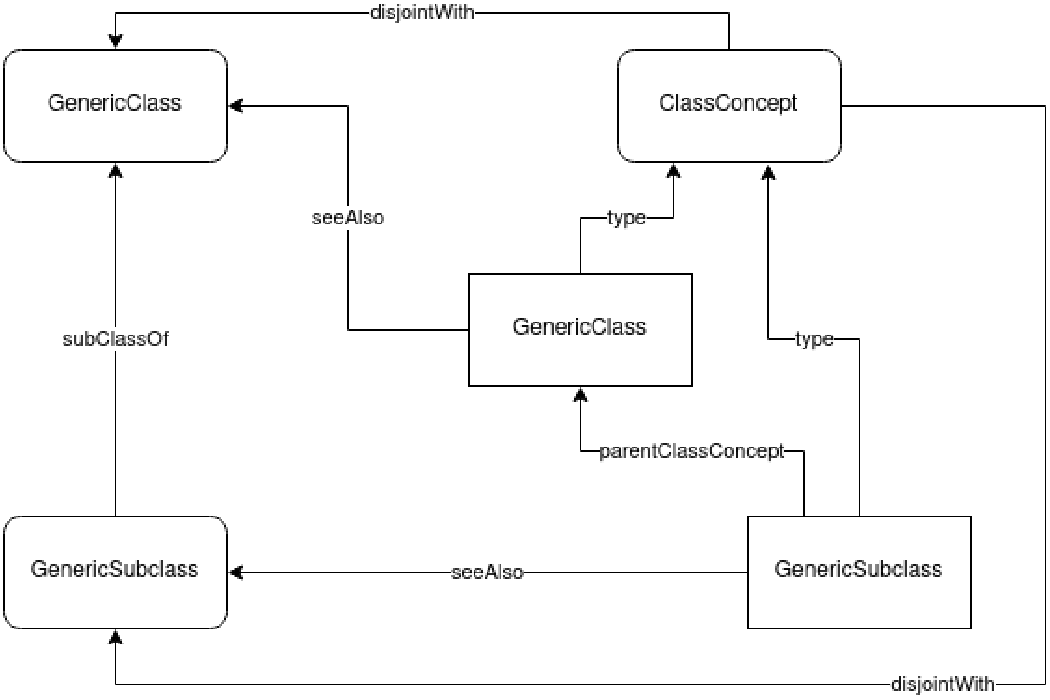

Fig.18. Generic implementation of intensional reification pattern

3.4 Exporting the graphs

The very last step in the generation of the result graph consists in the employment of **permanent IRIs**, which are needed also in the two input graphs since some nodes, such as `fred:materialise_1`, have only **temporary** IRIs. All the available graphs with permanent IRIs, both the input and the result ones, are available at this link. The graphs are all serialized in the `.owl` format.

Details about the deployment are described in the following section.

4 Deploy on a Docker container: Virtuoso, Lode, LodView

As a result, we provide the setup to download all the graphs within a Docker container, which hosts some services to allow **querying** the results (Virtuoso) and to ease the **browsing** of the ontologies (LODE and LodView).

The Docker container - when built and run following the instructions included in the README document - takes care of performing the following steps:

- downloading the dependency packages (needed to run the services)
- downloading the ontologies from the GitHub repository and loading them on Virtuoso
- launching the servers on dedicated ports

5 References

1. *Aldo Gangemi, Valentina Presutti, Diego Reforgiato Recupero, Andrea Giovanni Nuzzolese, Francesco Draicchio, Misael Mongiovì*. **"Semantic Web Machine Reading with FRED"**. Semantic Web Journal 8(6):873-893, 2017.
2. *Philipp Koehn*, **"Europarl: A Parallel Corpus for Statistical Machine Translation"**, MT Summit 2005
3. *Axel-Cyrille Ngonga Ngomo, Mohamed Ahmed Sherif, Kleanthi Georgala, Mofeed Hassan, Kevin Dreßler, Klaus Lyko, Daniel Obraczka, Tommaso Soru.* **"LIMES - A Framework for Link Discovery on the Semantic Web".** KI-Künstliche Intelligenz, German Journal of Artificial Intelligence - Organ des Fachbereichs "Künstliche Intelligenz" der Gesellschaft für Informatik e.V. 2021.
4. *Carl Boettiger*. **"rdflib: A high level wrapper around the redland package for common rdf applications (Version 0.1.0)**." Zenodo 2018.
5. *Presutti V., Daga E., Gangemi A., Blomqvist E.* **"eXtreme Design with Content Ontology Design Patterns."** WOP (2009).
6. *Musen M.A.* "**The Protégé project: A look back and a look forward."** AI Matters. Association of Computing Machinery Specific Interest Group in Artificial Intelligence, 1(4), June 2015.
7. *Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart*, **"Exploring network structure, dynamics, and function using NetworkX"**, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

8. *Honnibal et al.* **"spaCy: Industrial-strength Natural Language Processing in Python"**. Zenodo 2020.
9. *George A. Miller (1995).* **"WordNet: A Lexical Database for English."** Communications of the ACM Vol. 38, No. 11: 39-41.
10. *Christiane Fellbaum (1998, ed.)* **"WordNet: An Electronic Lexical Database."** Cambridge, MA: MIT Press.
11. *Bird, Steven, Edward Loper and Ewan Klein (2009),* **"Natural Language Processing with Python."** O'Reilly Media Inc.
12. *Jeffrey Pennington, Richard Socher, and Christopher D. Manning.* **"GloVe: Global Vectors for Word Representation."** 2014.
13. *Auer S., Bizer C., Kobilarov G., Lehmann J., Cyganiak R., Ives Z.* **"DBpedia: A Nucleus for a Web of Open Data."** In: Aberer K. et al. (eds) The Semantic Web. ISWC 2007, ASWC 2007. Lecture Notes in Computer Science, vol 4825. Springer, Berlin, Heidelberg. (2007)
14. *Gangemi A.* **"Ontology Design Patterns for Semantic Web Content."** In: Gil Y., Motta E., Benjamins V.R., Musen M.A. (eds) The Semantic Web – ISWC 2005. ISWC 2005. Lecture Notes in Computer Science, vol 3729. Springer, Berlin, Heidelberg.
15. *Merkel D*. **"Docker: lightweight linux containers for consistent development and deployment."** Linux journal. 2014;2014(239):2.
16. *Peroni S., Shotton D.M., Vitali F.* **"Making Ontology Documentation with LODE."** In Proceedings of the I-SEMANTICS 2012 Posters & Demonstrations Track, Graz, Austria, September 5-7, 2012, 63–67, 2012.
17. *Diego Valerio Camarda, Silvia Mazzini, and Alessandro Antonuccio.* **"LodLive, exploring the web of data."** In Proceedings of the 8th International Conference on Semantic Systems (I-SEMANTICS'12). ACM, New York, 197--200. (2012)
18. *Diego Valerio Camarda, Silvia Mazzini, and Alessandro Antonuccio.* **"LodView."** 2014
19. Virtuoso Open-Source Edition, link