



Universitat Politècnica de Catalunya

FACULTAT D'INFORMÀTICA DE BARCELONA

Master in Data Science

LABORATORY PROJECT ON KNOWLEDGE GRAPHS

Course project on Semantic Data Management

Project report

Arnau Arqué and Nora Herault

Grup 12

Professors: Oscar Romero and Md. Ataur Rahman

May 26, 2022

Contents

1	TBOX definition	3
1.1	Justification of choices	3
1.2	Other considerations	5
1.3	TBOX generation	5
2	ABOX definition	7
2.1	Data generator	7
2.2	ABOX generation	7
3	Creation of the final ontology	9
4	Querying the ontology	12
4.1	Find all Authors	12
4.2	Find all properties whose domain is Author	13
4.3	Find all properties whose domain is either Conference or Journal	13
4.4	Find all the papers written by a given author that where published in database conferences	14
5	Tools description	16
5.1	Data generator	16
5.2	TBOX and ABOX generator	16
6	Annex	17

List of Figures

4.1	Results obtained on the execution of 4.1 query.	12
4.2	Results obtained on the execution of 4.2 query.	13
4.3	Results obtained on the execution of 4.3 query.	14
4.4	Results obtained on the execution of 4.4 query.	15
6.1	Graphical representation of the TBOX.	19
6.2	Results obtained on the execution of 6.3 query.	21

List of Tables

3.1	Summary of the results obtained by executing the queries of Section 3.	11
6.1	Domain and range of each object property.	17
6.2	Domain and range of each datatype property.	18

Listings

2.1	Creation of Authors' individuals and its relations.	8
3.1	SPARQL query to obtain the number of classes.	9
3.2	SPARQL query to obtain the number of properties.	10
3.3	SPARQL query to obtain the number of instances of the poupulated classes. . . .	10
3.4	SPARQL query to obtain the number of triples using the main properties.	10
4.1	SPARQL query to obtain all the authors.	12
4.2	SPARQL query to obtain all properties whose domain is author.	13
4.3	SPARQL query to obtain all properties whose domain is either Conference or Journal.	14
4.4	SPARQL query to obtain the papers written by a given author that where published in database conferences.	15
6.1	Output provided by the TBOX and ABOX creation Java program.	17
6.2	Output provided by the data generator.	20
6.3	Alternative SPARQL query to obtain all properties whose domain is either Conference or Journal.	21

1 TBOX definition

In this section of the statement there is a description of a research publication domain. Based on this, we are asked to define a TBOX that includes the concepts that are defined in it. Since GraphDB does not provide any features to create a TBOX, we will have to find an alternative way to do it.

After discussing the best alternative for generating the TBOX, we decided to do it programmatically using the Jena API for Java. We believe that this API provides many tools (such as the Ontology API or RDF API) that will allow us, on the one hand, to generate the TBOX and, on the other hand, the ABOX. In addition, both authors of the project have previous experience with Java, which will greatly facilitate the generation of knowledge graph components.

1.1 Justification of choices

Despite having chosen to generate the TBOX using Jena, we will show its graphical representation first. This will make it easier for us to comment on our assumptions and justify our decisions. You can see the graphical representation of the TBOX in the figure 6.1¹.

As we can see, the graph shows the classes, relations (along with their labels) and subclass relations. To begin meeting the constraints, we've generated a **Paper** class that is made up of **title** attribute, which determines the title of the paper, **numberOfPages**, which determines its number of pages, and **issn**, which is an eight-digit paper ID code.

The statement states that a paper can be classified into four types. That's why we have defined four subclasses of **Paper**: **FullPaper**, **ShortPaper**, **DemoPaper**, and **Poster**. We have assumed that the subclasses **{Full,Short,Demo}Paper** have no distinctive attributes between the superclass, while **Poster** consists of two attributes **width** and **length** which determine the width and height of the poster.

As stated in the statement, a paper has an authorship. That's why we generated the **Author** class. To facilitate the justification of our choices, we will discuss later what attributes it has. To associate a paper with its author we have generated the relationship **hasAuthor**, with domain **Paper** and rank **Author**.

The author of a paper can present it to a venue, which can be either a conference or a journal. To represent this knowledge, we first generated a **Venue** class with a single **heading** attribute that represents its title or main concept. In addition, we have defined two subclasses of **Venue**: **Conference** and **Journal**. On the one hand, **Conference** consists of the **city**, **month**, and **year** attributes, representing the city, the month and the year in which the conference will be held, respectively. On the other hand, a **Journal** consists of the **edition** and **hasOnlineVersion** attributes, which indicate the edition of the journal and whether or not it has an online version.

¹Legend Extension: The values below the nodes and edge labels of the graph correspond to the number of attributes of the classes and relations, respectively.

As specified in the use case description, conferences can be classified into four distinct categories. To represent this knowledge, we have generated four subclasses of **Conference**: **Workshop**, **Symposium**, **ExpertGroup**, and **Regular**. Since all conference typologies work the same way, we did not consider it necessary to define custom attributes for each one.

Another fact to keep in mind is that conferences and journals are handled by chairs and editors, respectively. To represent this, we have defined two classes **Chair** and **Editor**. Next, we defined the relations **handlesConference** (with domain **Chair** and rank **Conference**) and **handlesJournal** (with domain **Editor** and rank **Journal**). Note that we could have defined a single relationship **handles** with domain **{Chair,Editor}** and rank **{Conference,Journal}**. However, we have decided to split it in two to simplify possible subsequent queries that require the use of these concepts.

As mentioned above, an author may submit his articles to a venue. To represent this fact we have generated the class **Submission**, consisting of the attributes **date** and **comment** which represent the date of the submission and the comments associated with it. In addition, a submission consists of a **published** attribute that indicates whether it has finally been published in the venue or not.

The domain description specifies that there must be a minimum of two reviews for each submission. To represent these reviews we have generated two new classes: **Review** and **Reviewer**. A review will consist of a **description** and a **decision**, while a reviewer has an associated **averageScore** ranging, which is a value from 1 to 10. In short, we will use the **Review** to represent the fact that a **Editor/Chair** assigns a **Reviewer** to evaluate a **Submission**.

To simplify the relations of **Review**, we decided to generate a class **Manager** and set it as a superclass of **Editor** and **Chair**. All of this will allow us to define the **assignedBy** relationship (with domain **Review** and rank **Manager**), which associates a review with the editor/chair assigned by the reviewer. To determine the reviewer of the review we generated the relation **hasReviewer** (with domain **Review** and rank **Reviewer**).

To finish defining the scope of submissions we have generated the following relations: **hasReview** (with domain **Submission** and rank **Review**) to associate a submission with your reviews, **hasPaper** (with domain **Submission** and range **Paper**) to associate the submission with the submitted paper, and finally **submittedTo** (with domain **Submission** and range **Venue**) to associate the submission with the venue where the paper is submitted.

The last concept we introduced is the area of a paper or a venue. To do this, we generated the class **Area** and the relations **paperRelatedTo** and **venueRelatedTo** (with domain **Paper** and **Venue**, respectively, and rank **Area**). We considered the option of generating a single **relatedTo** relationship, but in the end we chose to generate two for simplicity when performing subsequent queries. An area will consist of a single **field** attribute that designates its typology.

Note that during the previous paragraphs we did not mention which attributes are associated with the classes **Author**, **Reviewer** and **Manager** (as well as those of the subclasses **Editor** and **Chair**). It is not difficult to see that all the above classes are part of a person's concept. For this reason, as we can see in the graph, we have generated a class **Person** which is a superclass of **Author**, **Reviewer** and **Manager**. We considered a person to be defined by the **textttname** and **country** attributes.

Throughout the preceding paragraphs, for simplicity, we have not mentioned the domain and rank of the attributes of the classes represented. That is why we have generated two summary tables that specify the different domains and ranges of the relations between classes (table 6.1) and the attributes (table 6.2) defined.

1.2 Other considerations

It is not difficult to see that our representation of the TBOX (figure 6.1) does not in itself satisfy all the restrictions set out in the definition of the use case. We are aware of this, but, on the other hand, it is worth mentioning that we have done so for a reason. If we generated a scheme from which all constraints are met, we would obtain a set of relations and classes that is extremely complex and difficult to maintain and expand. That is why we have decided to create a simplified version, which is easier to maintain and to generate the corresponding ABOX from it.

All this does not mean, however, that the restrictions set out in the statement that are not directly satisfied in the TBOX should not be complied. It will therefore be necessary to take them into account when generating the different instances and relations of the ABOX. Next, we will list the extra restrictions that we will have to meet and, in later sections —when generating the ABOX— we will demonstrate that they are being met.

1. With the current scheme, there is no guarantee that the **Reviewers** of a **Submission** to a **Venue** of subtypes **Conference** and **Journal** will be assigned by **Managers** of subtype **Chair** and **Editor**, respectively. Therefore, this restriction must be met when generating or entering data.
2. It is also important to ensure that a **Paper** of subtype **Poster** can only be submitted to a **Venue** of subtype **Journal**.
3. The use case description specifies that one **Submission** must have at least two **Reviews**. Therefore, we will have to ensure this fact a posteriori.
4. We will consider a **Submission** to be published when most of its **Reviews** have the **decision** attribute set to "Accepted". Therefore, in these situations we must ensure that the **published** attribute of the submission takes the value "Yes".
5. The statement states that all venues and all papers must have at least one **Area** associated. Again, this restriction is not explicitly reflected in the TBOX. Therefore, we will make sure that there is always at least one **paperRelatedTo** and **venueRelatedTo** relationship for each **Paper** and **Venue**, respectively.

1.3 TBOX generation

At this point, we have discussed the concepts we have included in the TBOX and justified our decisions. To end with the definition of the TBOX, we will comment on how —specifically— we generated it.

As mentioned above, we decided to create the TBOX programmatically using the Jena APIs for Java. You can find the code in the Java **KnowledgeBaseCreation** project located in `./G11-Tools-ArqueHerault/` directory. The project was created using the IntelliJ IDEA IDE ² and includes a Run Configuration called **TBOX generation** which provides the parameters needed to run the code associated with the TBOX generation.

²<https://www.jetbrains.com/idea/>

The Java project mainly contains the class `KnowledgeBaseManager`. It will store the TBOX and the URIs that we will use to reference our classes, relations, and individuals. In fact, the class creation method requires the base URI to be used as a parameter. From there, it generates the URIs needed to generate the classes, relations, and individuals. In our case, we used the following URIs:

- `http://sdm/project/3/source/class` for classes.
- `http://sdm/project/3/source/objectProperty` for object properties (class relations).
- `http://sdm/project/3/source/datatypeProperty` for datatype properties (class attributes).
- `http://sdm/project/3/source/individual` for individuals.

The method that generates the TBOX is called `createTBOX`. To begin with, it creates a `OntModel` using the `ModelFactory.createOntologyModel` method in the Jena API. In the first instance, classes are generated using the `createClass` method. Subclass relations are then generated from the `addSubClass` method. Once these relations are defined, the reasoner will be able to infer the inverse relations (in this case, the superclass relation) and, therefore, we have decided not to define them explicitly.

Once the classes are generated, the process of generating object properties (*i.e.* class relations) begins. To do this, we used the `createObjectProperty` method. In addition, we thought it appropriate to define the domain and range of each relationship using the `addDomain` and `addRange` methods, respectively.

Finally, we define the datatype properties (*i.e.* the class attributes) using the `createDatatypeProperty` method and define, as we did with the object properties, the domain and rank of each attribute. Note that for datatype properties, the range will always be a literal value (*i.e.* `RDFS.Literal`, `XSD.integer`, etc.).

Once the TBOX is defined and generated, we save it to the file named `G11-B1-ArqueHerault.owl` located in the project's home directory.

2 ABOX definition

In this section we are asked to generate the ABOX associated with the use case that is exposed in the statement. To do this, we started by creating a data generator that creates the necessary instances and relations and at the same time satisfies all the established restrictions. Once we have generated the data, we have expanded the `KnowledgeBaseManager` mentioned in the previous section so that it now also generates the ABOX.

2.1 Data generator

As we introduced above, before generating the ABOX we required basic data that would allow us to instantiate all the classes and relations previously defined in the TBOX. That's why we made a Python program with this motivation. You can find its location and the instructions on how to run it in section 5.

This data generator implements the `KnowledgeBaseStorage` class, which is responsible for successively storing different individuals and relations and then depositing them in external CSV files. You can find the individuals generated in the `./data-generation/data/individuals/` directory and the relations in `./data-generation/data/relations/`. The program uses the parameters defined in the `parameters.py` file to create all individuals and relations.

The generator has been designed to comply with each and every one of the restrictions mentioned in 1.2. In addition to all this, the program offers explanatory outputs of the process, as well as a final summary of all the individuals and relations created. You can see an example of the output it produces in the listing 6.2.

To conclude, the program has been designed to be flexible and easy to use and update. For example, if we want to include new journals in our database, we only need to modify the `journals` variable in the file `parameters.py`. If we want to change the number of reviewers associated with each submission, we only need to change the `reviews_per_submission` variable of `KnowledgeBaseStorage`. All the code is commented properly to understand what needs to be changed in each situation.

2.2 ABOX generation

Once the data has been generated, we proceed to the generation of the ABOX. To do this, we have extended the program defined in section 1.3. More specifically, we have expanded the `KnowledgeBaseManager` class by adding a new `createABOX` method that, from the previously generated TBOX, instantiates classes and relations and stores them in a new model.

The `createABOX` method uses the previously generated TBOX model to instantiate individuals and relations. Let's look at an example:


```

1 List<List<String>> rows = getContents(ipath + "authors.csv");
2 for (List<String> row: rows) {
3     Individual ind = abox.createIndividual(individualSource + row.get(0),
4         tbbox.getOntClass(classSource + "Author"));
5     ind.addProperty(tbbox.getDatatypeProperty(datatypeSource + "name"),
6         row.get(1));
7     ind.addProperty(tbbox.getDatatypeProperty(datatypeSource + "country"),
8         row.get(2));
9     ind.addProperty(tbbox.getDatatypeProperty(datatypeSource + "hIndex"),
10        row.get(3));
11 }

```

Listing 2.1: Creation of Authors' individuals and its relations.

In the listing 2.1 we see how we read¹ the file `authors.csv` (created automatically by the data generator), which contains the information associated with all the authors we will use in this particular case. Next, we generate an individual for each row of the file (*i.e.* for each author) and specify that it belongs to the `Author` class of the previously generated TBOX. Finally, we add the author's attributes (*i.e.* datatype properties) using the `addProperty` method and, again, extracting their URI's type from the TBOX.

Using the TBOX model generated earlier to define the types of individuals and relations will allow us to not generate links between the TBOX and the ABOX later. We won't need any links because both TBOX and ABOX will share the same URIs.

Once the individuals are generated, we create the relations between them (*i.e.* The object properties). Again, we reused the URIs to avoid having to link the TBOX and ABOX later.

In order to generate the ABOX we have again provided an IntelliJ Run Configuration called `TBOX-ABOX generation`. The resulting ABOX is stored in the practice base directory in the `G11-B2-ArqueHerault.owl` file.

To conclude, we would like to point out that the code in the `KnowledgeBaseManager` class is—as well—well documented to make it easier to extend if necessary. The program also provides explanatory outputs that allow us to track the process. You can see an example of these outputs in the listing 6.1.

¹Using the `getContents` method we defined in class `KnowledgeBaseManager`.

3 Creation of the final ontology

In this section we are asked to generate the links between the TBOX and the ABOX previously generated. Once generated, the data must be uploaded to GraphDB. Next, we need to specify the inference regime entailment we are considering. Finally, we need to provide some data that summarizes the instances uploaded to the database.

In our particular case, as we have discussed in previous sections, it is not necessary that we generate any link between the TBOX and the ABOX, since we created the ABOX using the URIs of the classes and properties we defined in the TBOX. By having done this, GraphDB will have no trouble executing the required queries.

In the generation of our model, we used a model based on OWL-Full language, in-memory storage and RDFS inference regime entailment. In this way we have been able to use features provided by the OWL-Full language such as `owl:ObjectProperty` and `owl:DatatypeProperty`. In addition, the interference of the RDFS inference regime facilitates the generation of entailments from the sub-class and sub-property hierarchies.

A clear example of how we have taken advantage of the features of the inference regime we decided to use (RDFS) is that, among other things, we have been able to save the definition of some `rdf:type` in anticipation that the reasoner will be able to infer them when executing queries. In our case, as we have discussed in previous sections, we have had to define some subclass relations. For example, we have defined that both **Conference** and **Journal** classes are subclass of **Venue** class. However, given that we are using RDFS inference, we have not explicitly defined any inverse superclass relations.

Finally, we are asked to compute some basic statistics that show evidence that the data has been inserted correctly. To do this, we will show in each case the SPARQL queries we used to collect the data and comment on the results. In addition, you can see a summary table with the query results in the 3.1 table and the code of all queries in the `G11-B3-ArqueHerault.rq` file.

3.1 Number of classes

```
1 PREFIX owl: <http://www.w3.org/2002/07/owl#>
2 SELECT (COUNT(?class) AS ?numberOfClasses)
3 WHERE {
4     ?class a owl:Class
5 }
```

Listing 3.1: SPARQL query to obtain the number of classes.

The result of the query to get the number of classes was 21. If we look at the graph representing our TBOX (figure 6.1), we can see that, in fact, we have generated a total of 21 classes.

3.2 Number of properties

In this case, we are asked to calculate the number of properties. Unless otherwise specified, we will calculate how many object and datatype properties we have generated. You can see the query we used to obtain such result in listing 3.2.

```

1 PREFIX owl: <http://www.w3.org/2002/07/owl#>
2 PREFIX objProp: <http://sdm/project/3/source/objectProperty#>
3 PREFIX dtProp: <http://sdm/project/3/source/datatypeProperty#>
4 SELECT (COUNT(?property) AS ?numberOfProperties)
5 WHERE {
6     { ?property a owl:ObjectProperty }
7     UNION
8     { ?property a owl:DatatypeProperty }
9 }

```

Listing 3.2: SPARQL query to obtain the number of properties.

The result, as we can see in the table 3.1, is that we have generated a total of 31 properties. Again, if we look at the graph in the figure 6.1, we can see that there are a total of 10 relations between classes (*i.e.* object properties) and 21 attributes (*i.e.* datatype properties) distributed among the different classes. All in all, as we obtained from the query, we have a total of 31 properties.

3.3 Number of instances for the main classes

```

1 PREFIX owl: <http://www.w3.org/2002/07/owl#>
2 SELECT (COUNT(DISTINCT ?instance) AS ?numberOfInstances)
3 WHERE {
4     ?class a owl:Class.
5     ?instance a ?class
6 }

```

Listing 3.3: SPARQL query to obtain the number of instances of the populated classes.

The result has been that we have entered a total of 761 instances of the populated classes (which, in our case, are all the classes we have created in the TBOX). If we look at the output of the data generator (listing 6.2), we can see that the result of the query is correct and that we have indeed generated a total of 761 individuals.

3.4 Number of triples using the main properties

Finally, we are asked to calculate the number of triples used by the main properties. We will assume, again, that the main properties are both object and datatype properties. Let's see:

```

1 PREFIX owl: <http://www.w3.org/2002/07/owl#>
2 SELECT (COUNT(?subject) AS ?numberOfTriples)
3 WHERE {
4     { ?subject ?predicate ?object.
5         ?predicate a owl:ObjectProperty }
6     UNION
7     { ?subject ?predicate ?object.
8         ?predicate a owl:DatatypeProperty }
9 }

```

Listing 3.4: SPARQL query to obtain the number of triples using the main properties.

The total number of triples obtained is 3602. By a simple mathematical calculation, we can check that all the triples have been instantiated correctly. Ultimately, the number of triples is given by the expression $\text{triples} = \text{TriplesWithObjProps} + \text{TriplesWithDTProps}$. The `TriplesWithObjProps` can be obtained from the output provided by the data generator

(listing 6.2), that is, a total of **1655** triples that contain object properties. On the other hand, to calculate the number of triples involving a datatype properties it will be necessary to calculate:

$$\text{TriplesWithDatatypeProperties} = \forall c \in \text{Class}, \quad \sum \text{Ind}(c) \cdot \text{Attr}(c)$$

where c are the different classes we have generated and $\text{Ind}(c)$, Attr represent the number of populated individuals and the number of attributes of class c , respectively. So, using the data generator output (listing 6.2), we get¹:

$$\begin{aligned} \text{TriplesWithDTProps} &= \text{Ind}(\text{Person}) \cdot \text{Attr}(\text{Person}) + \text{Ind}(\text{Author}) \cdot \text{Attr}(\text{Author}) + \dots = \\ &= 100 \cdot 2 + 34 \cdot 1 + 43 \cdot 1 + 156 \cdot 3 + 18 \cdot 2 + 28 + 20 \cdot 3 + 8 \cdot 2 + 9 + 117 \cdot 3 + 351 \cdot 2 = \mathbf{1947} \end{aligned}$$

So we finally have $\text{triples} = 1655 + 1947 = \mathbf{3602}$, as we got in the execution of the query.

Table 3.1: Summary of the results obtained by executing the queries of Section 3.

Query	Concept	Result
3.1	Number of classes	21
3.2	Number of properties	31
3.3	Number of instances for the main classes	761
3.4	Number of triples using the main properties	3602

Source: Own elaboration.

¹The calculation $\text{Ind}(c) \cdot \text{Attr}(c)$ has been done for each class c generated (except for those that have no attributes, since $\text{Ind}(c) \cdot 0 = 0$) and in descending order to the one provided in the output 6.2.

4 Querying the ontology

At this point, we have generated the TBOX and the ABOX associated with the use case specified in the statement. Then, we uploaded the models to GraphDB and computed some basic statistics in order to monitor the data ingestion in the DB. In this section we are asked to run some SPARQL queries.

4.1 Find all Authors

In this section we are asked to find all the authors we have instantiated. You can find the code associated with the query in the listing 4.1. As you can see, in addition to the authors' URIs, we have also included their **name** attribute. We have put it this way so that the output is more explanatory. If we wanted to get only the URIs of the authors we would have to remove line 7 from the query and the **?name** variable.

```
1 PREFIX class: <http://sdm/project/3/source/class#>
2 PREFIX dtProp: <http://sdm/project/3/source/datatypeProperty#>
3 SELECT ?author ?name
4 WHERE {
5     ?author a class:Author;
6         dtprop:name ?name.
7 }
```

Listing 4.1: SPARQL query to obtain all the authors.

We can see the results obtained in the execution of this query in the figure 4.1. If we look at the output obtained by the data generator (listing 6.2), we can see that the number of authors matches the number of results obtained by the query (*i.e.* 34 authors).

Filter query results			Showing results from 1 to 34 of 34. Query took 0.1s, minutes ago.	
	author	↕	name	↕
1	http://sdm/project/3/source/individual#22		"Akira Otsuka"	
2	http://sdm/project/3/source/individual#23		"Pijush Samui"	
3	http://sdm/project/3/source/individual#11		"William Whyte"	
4	http://sdm/project/3/source/individual#9		"Yvo Desmedt"	
5	http://sdm/project/3/source/individual#5		"Volker Schillings"	

Figure 4.1: Results obtained on the execution of 4.1 query.

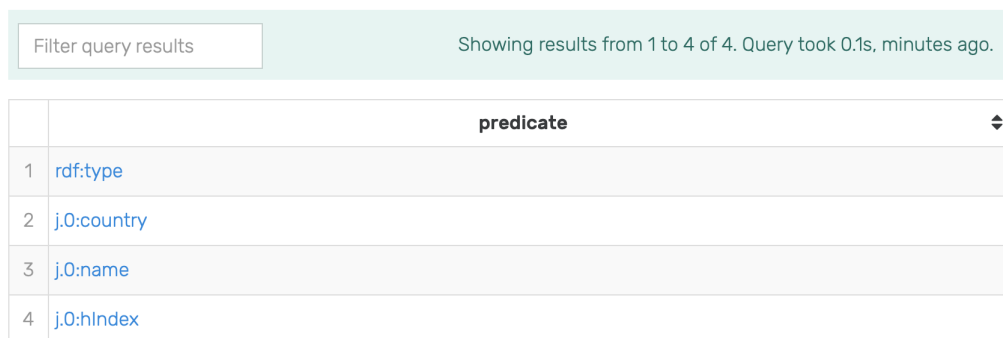
4.2 Find all properties whose domain is Author

If we look at the TBOX schema we generated (figure 6.1), we can see that there is no relation between classes which has the **Author** class as a domain. However, the class does have attributes, which are indeed properties (more specifically, datatype properties). So let's check the result. The query code is in the listing 4.2.

```
1 PREFIX class:<http://sdm/project/3/source/class#>
2 PREFIX owl: <http://www.w3.org/2002/07/owl#>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
5 SELECT DISTINCT ?predicate
6 WHERE {
7     ?subject ?predicate ?object;
8         a class:Author
9 }
```

Listing 4.2: SPARQL query to obtain all properties whose domain is author.

We can see the results of the query in the figure 4.2. As expected, the properties obtained were **name**, **country**, **hIndex** and **rdf:type**. There are no object properties in the results because there is no class relation with **Author** as the domain.



	predicate
1	rdf:type
2	j0:country
3	j0:name
4	j0:hIndex

Figure 4.2: Results obtained on the execution of 4.2 query.

Note that the result contains the datatype properties **name** and **country**, which are programmatically defined on the **Person** class. This behavior of the reasoner is correct since we have declared that **Author** is *subclass* of **Person**.

4.3 Find all properties whose domain is either Conference or Journal

Again, we are asked to retrieve properties. Now, however, these must have either **Conference** or **Journal** as its domain. We might think that the structure of the query should necessarily have a **UNION** between properties with domain **Conference** and those with domain **Journal** (listing 6.3). However, if we look at the structure of our TBOX (figure 6.1), we see that **Conference** and **Journal** are the only subclasses of **Venue**. Therefore, the behavior described above will be equivalent to finding properties with domain **Venue**. The query code is in the listing 4.3.

```

1 PREFIX class: <http://sdm/project/3/source/class#>
2 SELECT DISTINCT ?property
3 WHERE {
4     ?instance ?property ?range;
5         a class:Venue
6 }

```

Listing 4.3: SPARQL query to obtain all properties whose domain is either Conference or Journal.

The query results can be found in Figure 4.3. As expected, we have got all the properties (object and datatype properties) of **Venue**, **Conference**, and **Journal**. If we run the alternative query (listing 6.3), we see that, indeed, the results (figure 6.2) are equivalent to those obtained in this section.

Filter query results		Showing results from 1 to 8 of 8. Query took 0.1s, moments ago.	
		property	↕
1		rdf:type	
2		j1:venueRelatedTo	
3		j0:hasOnlineVersion	
4		j0:edition	
5		j0:heading	
6		j0:year	
7		j0:city	
8		j0:month	

Figure 4.3: Results obtained on the execution of 4.3 query.

4.4 Find all the papers written by a given author that where published in database conferences

To conclude, we are asked to find all the papers written by a given author that where published in database conferences. You can find the query code in the listing 4.4. We have organized the query so that it provides us with extra information (in addition to the URIs) so that the result is more readable.

As we can see, we have made an initial **BIND** of some control variables that will help us meet the constraints of the query. We want to mention that we have structured the query in this way to facilitate the modification of the query restrictions. However, we could have directly replaced the values of the binded variables directly where we refer to them in the **WHERE** clause.

Once the query was executed, we obtained the results shown in the figure 4.4. As can be seen, we have obtained a single paper with title «*Analysis for REPERA: A Hybrid Data Protection Mechanism in Distributed Environment.*» written by the selected author (William Whyte). It was published in the conference heading «*Web Application Security*» which actually belongs to the area with field *Database*.

```

1 PREFIX ind: <http://sdm/project/3/source/individual#>
2 PREFIX class: <http://sdm/project/3/source/class#>
3 PREFIX dtProp: <http://sdm/project/3/source/datatypeProperty#>
4 PREFIX objProp: <http://sdm/project/3/source/objectProperty#>
5 SELECT ?paper ?paperTitle ?author ?authorName ?submission ?conf ?confHeading
   ?area ?field
6 WHERE {
7     BIND(ind:11 AS ?author) # Modify to change the given Author
8     BIND("Database" AS ?field) # Modify to change the field of the Area
9     BIND("Yes" AS ?published) # Modify to change the published status
10    ?paper a class:Paper;
11         objProp:hasAuthor ?author;
12         dtProp:title ?paperTitle.
13    ?author dtProp:name ?authorName.
14    ?submission a class:Submission;
15         objProp:hasPaper ?paper;
16         objProp:submittedTo ?conf;
17         dtProp:published ?published.
18    ?conf a class:Conference;
19         objProp:venueRelatedTo ?area;
20         dtProp:heading ?confHeading.
21    ?area dtProp:field ?field
22 }

```

Listing 4.4: SPARQL query to obtain the papers written by a given author that where published in database conferences.

Filter query results		Showing results from 1 to 1 of 1. Query took 0.1s, moments ago.							
	paper ⇅	paperTitle ⇅	author ⇅	authorName⇅	submission ⇅	conf ⇅	confHeading⇅	area ⇅	field ⇅
1	http://sdm/project/3/source/individual#193	"Analysis for REPERA: A Hybrid Data Protection Mechanism in Distributed Environment."	http://sdm/project/3/source/individual#11	"William Whyte"	http://sdm/project/3/source/individual#569	http://sdm/project/3/source/individual#279	"Web Application Security"	http://sdm/project/3/source/individual#100	"Database"

Figure 4.4: Results obtained on the execution of 4.4 query.

5 Tools description

In this section we want to make a brief description of how the tools used to carry out this project are organized and, if necessary, their execution instructions. All the tools referenced in this section can be found in the `./G11-Tools-ArqueHerault` directory.

5.1 Data generator

The data generator is a Python program created for the purpose of creating the necessary data to carry out this project (individuals and relations) in CSV files. Subsequently, this data will be used in the generation of the ABOX. You can find the program in the `./G11-Tools-ArqueHerault/data-generation/` directory. To run it, enter the following command line from the directory specified above:

```
> python3 data-generator.py
```

As the program runs, it will provide a series of messages indicating its status. If the process completes successfully, you will be able to see a summary of the instances and relations generated (see listing 6.2).

5.2 TBOX and ABOX generator

The generation of the TBOX and the ABOX is implemented with a Java program. You can find the source code in the `./G11-Tools-ArqueHerault/KnowledgeBaseCreation/` directory. More specifically, the program implements a single class called `KnowledgeBaseManager`, the source code of which is located in the `src` subdirectory of the project. To run the program, follow these steps:

1. Open IntelliJ IDEA.
2. Go to **File** -> **Open**, navigate to the root directory of the project and, finally, select **Open**.
3. Go to **File** -> **Project Structure** -> **Project** and set the SDK to version 11.
4. Select the appropriate **Run Configuration** from the top-right corner.
 - a) If you want to generate only the TBOX, choose the **TBOX generation** configuration.
 - b) If you want to generate both the TBOX and the ABOX, choose the **TBOX-ABOX generation** configuration¹.
5. Click on the **Run** button next to the **Run configuration** picker.

As the program runs, a series of messages will be provided indicating its status. If the process completes successfully, a message will be provided indicating the location of the generated files (see listing 6.1).

¹Note: The **TBOX-ABOX generation** configuration requires that the data generation program (see section 5.1) is executed beforehand.

6 Annex

Table 6.1: Domain and range of each object property.

Domain	Object Property	Range
Submission	submittedTo	Venue
	hasPaper	Paper
	hasReview	Review
Review	hasReviewer	Reviewer
	assignedBy	Manager
Paper	hasAuthor	Author
	paperRelatedTo	Area
Venue	venueRelatedTo	Area
Editor	handlesJournal	Journal
Chair	handlesConference	Conference

Source: Own elaboration.

```

-----
-- SDM Project 3
-- Authors: Arnau Arque and Nora Herault
-----
-- TBOX and ABOX creation program
-----

[Creating the TBOX...Success!]
[Creating the ABOX...Success!]
[TBOX saved at '..../G11-B1-ArqueHerault.owl']
[ABOX saved at '..../G11-B2-ArqueHerault.owl']

Process finished with exit code 0

```

Listing 6.1: Output provided by the TBOX and ABOX creation Java program.

Table 6.2: Domain and range of each datatype property.

Domain	Datatype Property	Range
Review	description	RDFS.Literal
	decision	RDFS.Literal
Paper	title	RDFS.Literal
	numberOfPages	XSD.integer
	issn	RDFS.Literal
Poster	width	XSD.integer
	length	XSD.integer
Submission	comment	RDFS.Literal
	date	XSD.date
	published	RDFS.Literal
Area	field	RDFS.Literal
Venue	heading	RDFS.Literal
Conference	city	RDFS.Literal
	month	RDFS.Literal
	year	XSD.integer
Journal	edition	XSD.integer
	hasOnlineVersion	RDFS.Literal
Person	name	RDFS.Literal
	country	RDFS.Literal
Author	hIndex	XSD.integer
Reviewer	averageScore	XSD.integer

Source: Own elaboration.

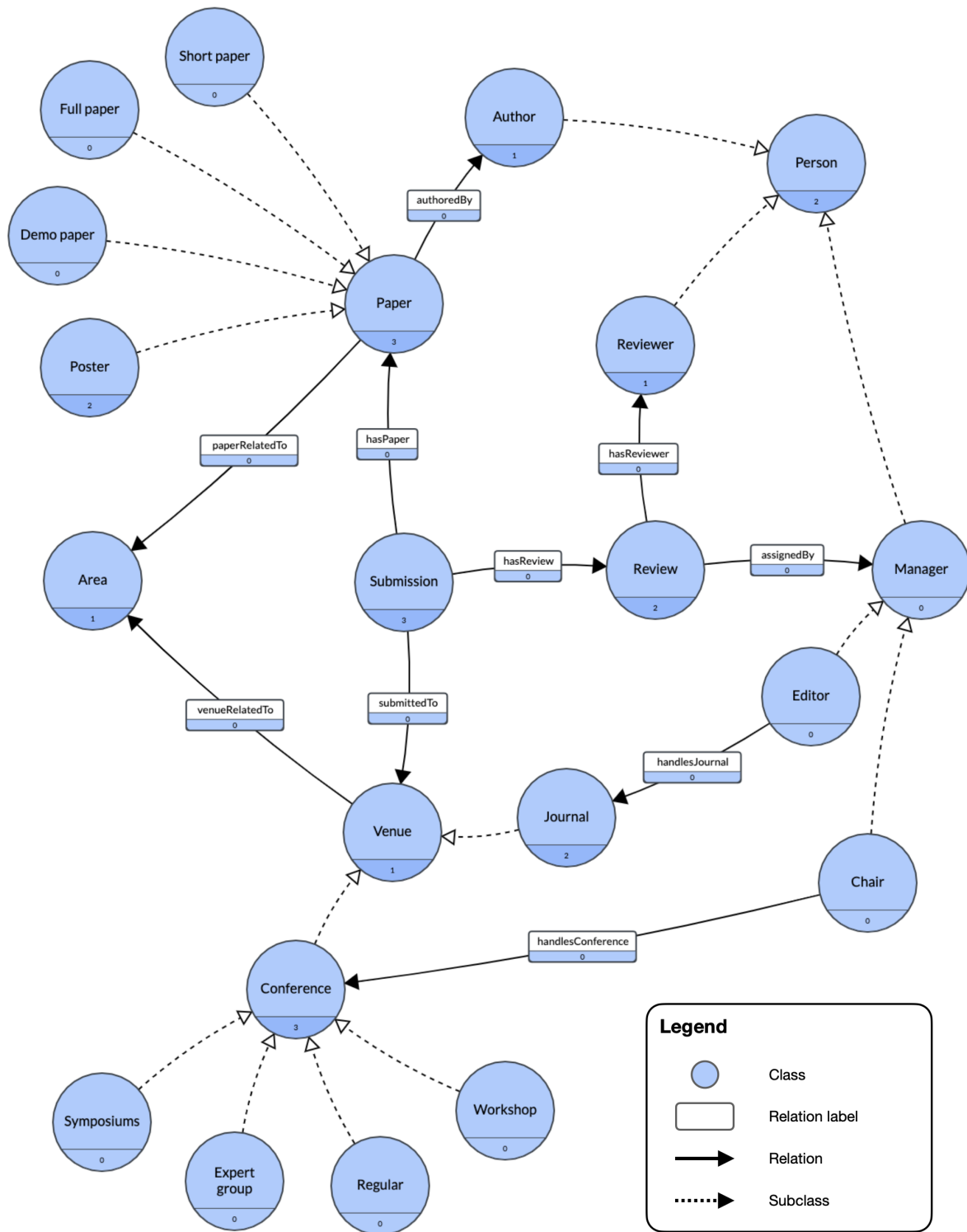


Figure 6.1: Graphical representation of the TBOX.

```

$ python3 data-generator.py

-----
-- SDM Project 3
-- Authors: Arnau Arque and Nora Herault
-----
-- Data generation program
-----

[Generating instances...]
[Generating relations...]
[Succeed]

[Summary of INDIVIDUALS] 761 individuals generated:
  [Persons] (100)
  |   34 authors
  |   43 reviewers
  |   [Managers] (23)
  |   |   12 editors
  |   |   11 chairs
  [Papers] (156)
  |   18 posters
  |   26 demo papers
  |   10 short papers
  |   102 full papers
  [Venues] (28)
  |   [Conferences] (20)
  |   |   5 workshops
  |   |   6 symposiums
  |   |   3 expert groups
  |   |   6 regulars
  |   8 journals
  9 areas
  117 submissions
  351 reviews

[Summary of RELATIONS] 1655 relations generated:
  351 assignedBy
  20 handlesConference
  8 handlesJournal
  156 hasAuthor
  117 hasPaper
  351 hasReview
  351 hasReviewer
  156 paperRelatedTo
  117 submittedTo
  28 venueRelatedTo

```

Listing 6.2: Output provided by the data generator.

```

1 PREFIX class: <http://sdm/project/3/source/class#>
2 SELECT DISTINCT ?property
3 WHERE {
4     { ?instance1 ?property ?range1;
5       a class:Conference }
6     UNION
7     { ?instance2 ?property ?range2;
8       a class:Journal }
9 }

```

Listing 6.3: Alternative SPARQL query to obtain all properties whose domain is either Conference or Journal.

Filter query results

Showing results from 1 to 8 of 8. Query took 0.1s, moments ago.

	property	
1	rdf:type	
2	j.1:venueRelatedTo	
3	j.0:heading	
4	j.0:year	
5	j.0:city	
6	j.0:month	
7	j.0:hasOnlineVersion	
8	j.0:edition	

Figure 6.2: Results obtained on the execution of 6.3 query.