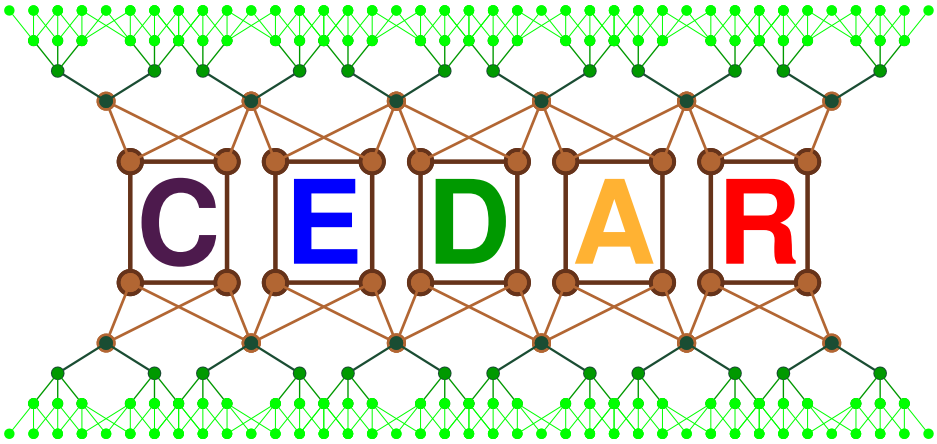


Technical Report Number 14

GAIA

An OWL-based Generic RDF Instance Generator

Tanguy Raynaud, Rafiqul Haque, Samir Amir, Mohand-Saïd Hacid

November 2014

Publication Note

Corresponding Author:

$\mathcal{CEDAR}$ Project

LIRIS - UFR d'Informatique
Université Claude Bernard Lyon 1
43, boulevard du 11 Novembre 1918
69622 Villeurbanne cedex
France

Phone: +33 (0)6 28 07 34 77
Email: cedar@liris.cnrs.fr

This work was carried out as part of the $\mathcal{CEDAR}$ Project (Constraint Event-Driven Automated Reasoning) under the *Agence Nationale de la Recherche* (ANR) Chair of Excellence grant N^o **ANR-12-CHEX-0003-01** at the *Université Claude Bernard Lyon 1* (UCBL).

CEDAR **Technical Report Number 14**

GAIA
An OWL-based Generic RDF Instance Generator

Tanguy Raynaud, Rafiqul Haque, Samir Amir, Mohand-Saïd Hacid
cedar@cedar.liris.fr

November 2014

Abstract

An RDF (Resource Description Framework) instance generator produces RDF triples by complying with an ontology that defines classes, subclasses, relations, and constraints. There are many instance generators which rely on Web Ontology Language (OWL) meaning that these generators can read only the ontologies which are written in OWL. However, the existing generators are locked-in to a specific ontology, which means the generators can read only a specific ontology. For instance, the LUBM generator can read only the LUBM ontology, which is clearly a limitation as it is not able to read any other ontology such as *biomodel ontology*. This promotes the need for a generic RDF instance generator that is able to read and parse any ontology written in OWL. In this technical report, we describe a generic RDF instance generator. We develop this generator to enable users to use their own ontology to generate RDF triples which can be used to meet their specific needs.

Keywords: OWL, RDF, Semantic Web, Ontology, Taxonomy, Instances generation

Table of Contents

1 Introduction

Generalization and *specialization* are two distinct styles of designing software applications. Generalization enhances the *usage* of a solution whereas specialised solutions cover specific contexts. However, developing generic solutions is enormously challenging; in fact, it is far more challenging than the specific ones. Specifically, developing generic functional features of a solution is an intricate problem. Thus, such solutions are not readily available. For example, there is no generic solution which can generate Resource Description Framework (RDF) instances from any ontology even if there is no technological heterogeneity.[1] An ontology can be defined as a formal document which contains knowledge. The document is composed of

- *concepts (classes)* - which a representation of a thing, *data properties*,
- *individuals* - which is an instance of a concept,
- *relations* - which is a link between a class with one another. Furthermore, properties are classified into two:
- *data properties*, and
- *object properties* (a link to another object).

Every ontology has a specific definition of the hierarchical structure of concepts. Additionally, the definition of the axioms of different ontologies varies. Therefore, two ontologies, although developed using the same language such as Web Ontology Language (OWL), a generator cannot read or parse. [2] For example, LUBM is an RDF triple generator which can read only a specific schema called `univ-bench.owl`. [3] Any other ontology if supplied to this generator, it will produce an exception. Such specialized instance generators are bound to the definition of specific classes (concepts), subclasses (objects), and attributes contained in a given schema - which is a limitation of the state of the art.

In this research work, we take an initiative to overcome this limitation. we develop a OWL–based generic RDF triple generator which enables to loading any ontology schema written in OWL and to producing RDF instances by complying with the ontology. We called the generator *GAIA* (Automatic Instance Generator for Abox). We strictly rely on OWL because we found that almost all ontologies that are currently available are defined/developed using this language. The wide adoption of this language influenced our decision. This technical report provides the detail of the implementation of our generator.

This technical report is organized as follows. In section 2 we present the motivation behind this work. Section 3 presents a high-level architecture of the generator. Section 4 provides the detail of the development of the generator. We conducted a few tests which are reported in Section 4. We draw a conclusion of this research in Section 6.

[1] http://www.w3.org/RDF/
[2] http://www.w3.org/TR/owl-features/
[3] http://swat.cse.lehigh.edu/projects/lubm/

2 Motivation

As part of the CEDAR project, one of our goals was generating a large-scale RDF data set from NCBI ontology. We found two well-known generators: LUBM and BSBM. We first tried with the LUBM generator. We loaded ncbi schema "ncbi.owl" onto LUBM engine and then we start the generator. However, it produces an error *unknown format* while parsing the concept and properties. We encountered the similar error during our experiment with BSBM. [4]

After studying both generators, we conclude that they are functionally rich and can efficiently be used as a black-box, yet they are locked-in to specific ontology schemas. For example, LUBM is locked-in to `univ-bench.owl` ontology. This motivated us towards the development of a generic OWL-based generator.

3 Solution Architecture & Life cycle

In this section, we present a high-level architecture of our generator. Additionally, we briefly describe the instance-generation life cycle.

The generator consists of four components: a *loader*, a *parser*, an *extractor*, and a *generator*. Figure 1 shows these components.

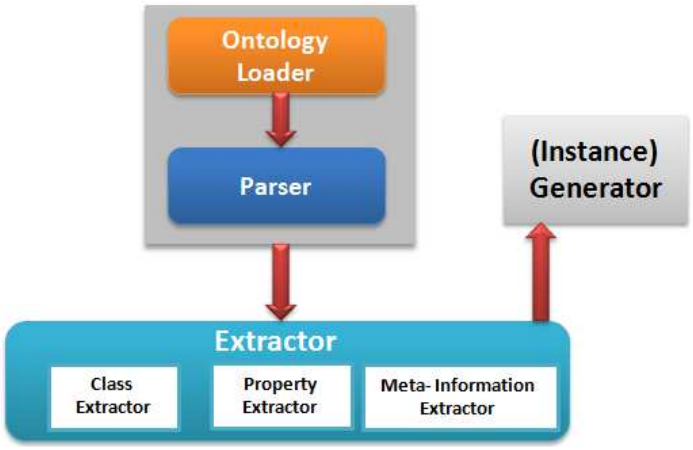


Figure 1: The Solution Architecture the Generator

- *The Loader*: It loads ontology schemas onto memory.
- *The Parser*: It parses the given schemas.
- *The Extractor*: It extracts classes, properties, and information. The extractor extracts two types of properties: *data properties* and *object properties*.
- *The Generator*: It produces instances, store them in buffer, and then write them in file.

[4] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/

The implementation of these components are discussed in section 4.

The RDF instance generation life cycle comprises six steps which are listed below.

- **STEP 1**: An ontology schema is loaded and parsed in this step.
- **STEP 2**: The classes contained in the schema are extracted.
- **STEP 3**: The data properties extracted in this step. This step produces a list containing all data properties referenced in the ontology. Also, for each of property, it extracts domains and range(which can be *Integer*, *String*, *Double*, *Boolean* or *Float*.) Note that, a property can have only one range but many domains.
- **STEP 4**: The object properties are extracted in this step.
- **STEP 5**: In this step, the generator extracts (meta-)information about the concept hierarchy. More specifically, this step identifies the relationships between concepts, sub–concepts, and super–concepts. This step produces a *java object* called `OwlOntology` which contains all information extracted by the API.
- **STEP 6**: This step produces the instances of the classes. The instances are triples that are composed of subjects, predicates, and objects.

4 Development of the Generator

In this section, we provide the implementation detail of GAIA. We used Java development environment. We used the OWL API for loading and parsing ontology schemas. The API enables the generator to extract different classes and properties contained in a given schema. In this research work, we concentrate more on developing the extractor and the generator. We describe the implementation of these components in the following subsections.

4.1 The Extractor

The extractor consists of three components: *the class extractor*, *the property extractor*, and *the information extractor*. Below, we provide a description of these extractors.

Class Extractor

For each concept of a given ontology, the class extractor creates an internal object called `TClass` which is a kind of container. A `TClass` stores information about the concepts, their relations with child nodes(sub-concepts), and properties of the concepts contained in ontologies. Figure 2 shows the structure of the `TClass`.

Also, the class extractor discovers the hierarchical relation between a concept and its sub–concepts and super–concepts. The pseudocode of the class extractor implementation is shown as Algorithm 1.

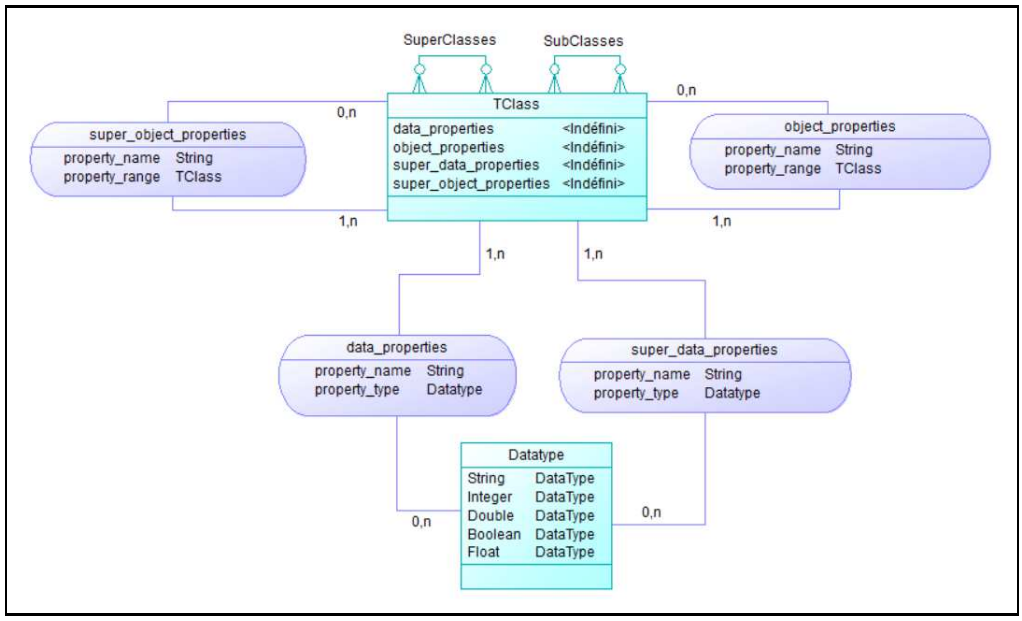


Figure 2: The conceptual diagram of TClass

Data: An OWLOntology(ont) object from OWLAPI
Result: A Map<String,TClass> ($map\_classes$) containing all the classes of the ontology

initialization;
foreach *$oClass$ in ont.getClasses()* **do**
  if *not $map\_classes$.contains($oClass$.getIRI())* **then**
    $map\_classes$.insert($oClass$.getIRI(), new TClass($oClass$.getIRI()));
  end
  foreach *$subClass$ in $oClass$.getSubClasses()* **do**
    if *not $map\_classes$.contain($subClass$.getIRI())* **then**
      $map\_classes$.insert($subClass$.getIRI(), new TClass($oClass$.getIRI()));
    end
    $subClass$.addSuperClass($oClass$);
    $oClass$.addSubClass($subClass$);
  end
end

Algorithm 1: Extracting classes

Data Property Extractor

The data properties cannot be extracted directly. The extractor first finds the domain and ranges of a data property and then uses the domain to extract the data properties. The data properties are stored in corresponding objects ('`TClass`'). In order to store the data properties, each object ('`TClass`') uses a hash-table of type HashMap `HashMap<String, DataType>`, where where the key of type String is the name of the property, and the value of type DataType is the property. The implementation of the data property extractor is shown as Algorithm 2.

Data: An OWLOntology(ont) object from OWLAPI, and a Map of TClass($map\_classes$) initialized in the previous step
Result: The data properties from the initial ontology are added to their respective TClass

foreach $oDataProp$ *in* ont.*getDataProperties()* **do**
  Datatype $type$ = new Datatype($oDataProp$.getRange());
  String $property\_name$ = $oDataProp$.getName();
  foreach $oDomain$ *in* $oDataProp$.*getDomains()* **do**
    TClass $tDomain$ = $map\_classes$.find($oDomain$.getIRI());
    $tDomain$.addDataProperty($poperty\_name, type$);
    $subClass$.addSuperClass($oClass$);
    $oClass$.addSubClass($subClass$);
  end
end

Algorithm 2: Extracting Data properties

Object Property Extractor

The proposed generator relies on the OWL API to extract object properties with domains and ranges.[5] It is worth noting that a range of an object property can be a concept contained in an ontology.

The object property extractor extracts these object properties through an association with the corresponding domains and ranges. This properties are then stored in `TClass` with `HashMap<String,TClass>`, where key denotes property name, and the value is a reference on the `TClass` used as range.

Information Extractor

An ontology represents the hierarchical subclass relations among classes of the real-world "thing". The information of these relations are critical to create instances. Thus, we developed an information extractor. It extracts information of relation between classes and their subclasses and superclasses. In addition, it extracts information about

[5] http://owlapi.sourceforge.net/

subproperties and superproperties. Extracting this information is straightforward because in ontology, the properties of a class are inherited by its subclasses; the inheritance relation enables the information extractor to extract the properties of superclasses (we call them *superproperties* in this report) contained in a ontology. In order to get all superproperties, we implement a recursive algorithm which is shown in Algorithm 3.

Input: a TClass($tClass$), Map<String,Datatype> $map\_super\_data\_properties$,
Map<String,TClass> $map\_super\_object\_properties$
Result: The classes are recursively explored to extract all super properties

if *not* $tClass$*.isExplored()* **then**
  foreach $superClass$ *in* $tClass$*.getSuperClasses()* **do**
    recursiveExploreSuperNodes($superClass$,
    $tClass$.getSuperDataProperties(), $tClass$.getSuperObjectProperties());
    $tClass$.setExplored();
  end
end
$map\_super\_data\_properties$.addMany($tClass$.getSuperDataProperties());
$map\_super\_data\_properties$.addMany($tClass$.getDataProperties());
$map\_super\_object\_properties$.addMany($tClass$.getSuperObjectProperties());
$map\_super\_object\_properties$.addMany($tClass$.getObjectProperties());

Algorithm 3: Recursive explore super nodes

At the end of object property extraction process, the generator stores all information which is used later during the instance generation phase.

4.2 The Instance Generator

The instances generator has been implemented in two successive incremental versions. The first version generates instances with *data properties* only. The subsequent version produces instances with both data and object properties. In this section, we provide the implementation detail of both versions.

4.2.1 The Initial Version

The first version of the generator can be used in a special case. To be specific, it extracts instances from the ontologies which represent *taxonomies* only. A taxonomy is a specific type of ontology, where relations between concepts and their child nodes are defined hierarchically without defining any complex property such as *symmetric property* of concepts. A taxonomy is an ontology without properties. Figure 3 provides an example of a taxonomy.

During instance generation from such ontologies, the data properties of classes must be propagated to all of its child nodes (sub-classes) in the hierarchy. The data properties can be any of the following types: `Boolean`, `Double`, `Float`, `Integer`, and

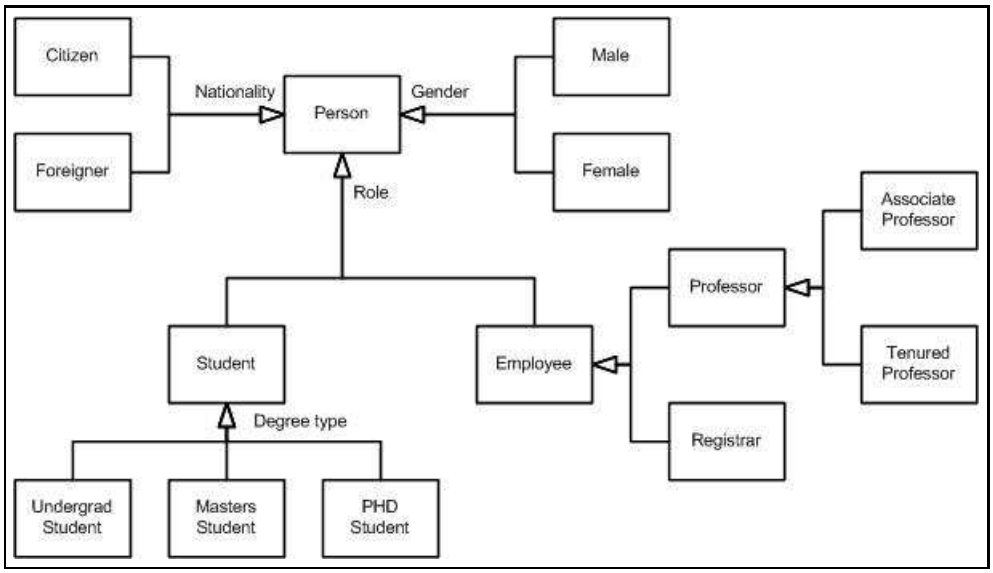


Figure 3: An example of taxonomy

`String`.

Instances are generated in two steps. First, the generator copies the given ontology file (loaded onto the main memory) and writes instances at the end of that file. Note that, in some cases, the generator creates a new file and adds instances there instead of adding them to the ontology file. Then, in the second step, the generator writes the instances to the file stored in secondary storage. This approach frees main memory significantly and thus, prevents memory overflow and optimizes the performance.

In the extraction phase, the generator defines an association between concepts and the domains of data properties inherited from the super-classes. This information are stored in `TClass`. This enables the generator to be more efficient. more efficient because, for any concept, the generator creates its instances without the need to search its super-attributes. Furthermore, our generator uses a buffer instead of accessing secondary storage devices (hard disks), which enhances its performance. Once the buffer exceeds a predetermined storage capacity, its contents are moved to a file using the `WRITE()` function. Then the buffer is automatically reset to empty state by the generator. The pseudocode in Algorithm 4 demonstrates the implementation of the initial version and how it works. It is worth noting that this algorithm is integrated into the final version.

Instance Generation Steps

Below, we provide the list of steps which show how instances are generated using the initial version:

- The users should specify the number of instances to be generated for each concept in the ontology. The value on the variable 'number of instances' can be a

Input : The input file ($iFile$), the output file($oFile$), the numbers of instance to generate (min and max)
Output: The number of instances generated

initialization;
$oFile$ = copyFile($iFile$); **foreach** $tClass$ *in* $map\_classes$ **do**
  int $numberInstances$ = min;
  if $max > min$ **then**
    $numberInstances$ = randomInt(min, max);
  end
  for *int* i = *0 to* $numberInstances$ **do**
    Individual ind = new Individual($tClass$.getIRI);
    foreach $data\_prop$ *in* $tClass$.*getDataProperties()* **do**
      ind.addRandomTypedAttribute($data\_prop$.getPropertyName(), $data\_prop$.getPropertyType());
    end
  end
  $buffer$.append(ind.getRDF());
  if $buffer$.*size()* $> MAX\_BUFFER\_SIZE$ **then**
    $oFile$.append($buffer$);
  end
end

Algorithm 4: First version : only data properties

single value or a *range*.

- The generator creates an temporary object called `Individual` for each concept.
- Then, RDF instances are generated using the individual. The name on the individual is defined by concatenating the name of the corresponding classes and an unique instance identifier (ex: `www.ontologyIRI/myclass_instance_1`). The unique identifier is incremented for each new instance.
- Then, a random value is chosen for all data properties of the instance.
- Finally, the instances are moved from the memory to a file.

Multithreading

Performance is the most critical factor of this way of proceeding. This solution for instance generation focuses on maximizing performance. Multithreading allows a better use of resources than sequential processing. To enhance performances of the generator, we modified the above algorithm to distribute the tasks among many threads. We implemented three distinct multithreading models: *Single-unit Threading*, *Dual-unit Threading*, and *Mass-unit Threading*. Figure 4 depicts these approaches.

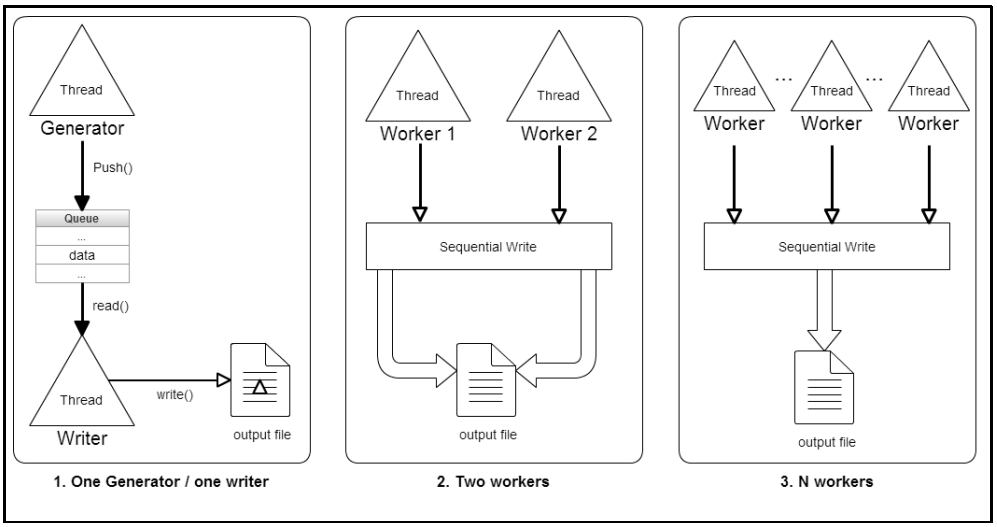


Figure 4: The different multi-thread implementations

A. Single-unit Threading

This model consists of two connected threads which perform complementary but distinct tasks, which is why it is called *Single-unit Threading*. In this model, in a cycle, each thread performs one unit-task only: either generation of instances or writing instances in files. We call the first thread the *generator thread* since it generates instances and puts them in a queue. The second thread is the *writer thread* which dequeues instances, and writes them into the output file.

After observation of actual runs, single-unit threading's performance was not as good as that of the single-thread's. This was not what we had expected. In fact, the *Single-unit Threading* approach consumes more time than the single-thread generator.

B. Dual-unit Threading

The *Dual-unit Threading* model is a multi-threading approach where each of two workers performs both *generation* and *writing* simultaneously. In this approach, the number of thread is limited to two. In order to avoid racing condition (for reserving the main memory) we implemented the *mutual-exclusion* technique using the *mutex* API. The API enables the generator to prevent the writer threads from competing for memory access. In other words, when a thread uses the output buffer, the other threads waits.

C. Mass-unit Threading

The *Mass-unit Threading* model enables using number of threads for performing the generation and writing of instances. It is essentially the extended version of the *Dual-unit Threading* model. We expected the *Mass-unit Threading* approach to supersede

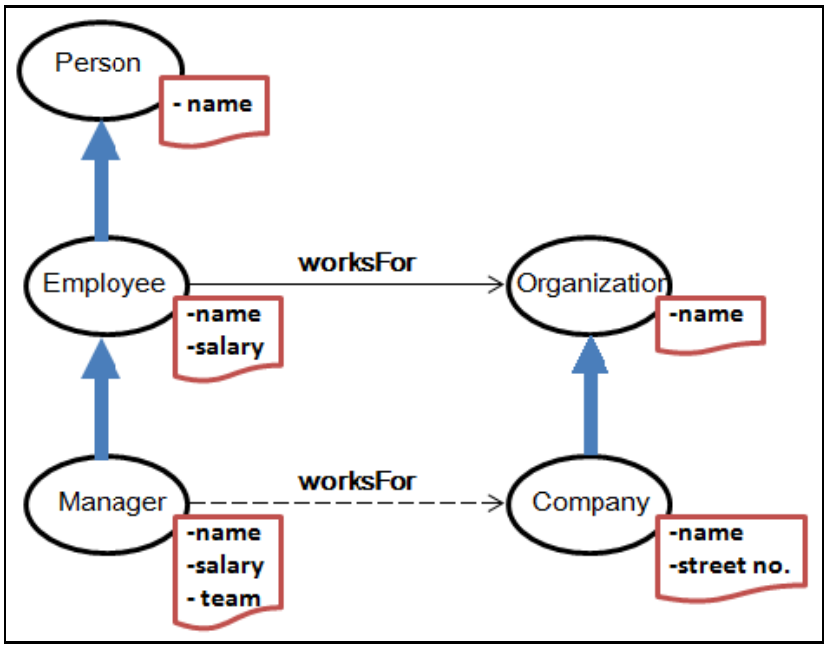


Figure 5: Inheritance of the properties

the performance of of the two other models. However, the experiments showed otherwise. The performance of mass-unit threading model was the worst. According to our observation, when running often, the threads the threads consume a considerable amount of time waiting in the queue for the running thread to release the *mutex*. This is for the case when all threads share a single *mutex*.

Based on our observation, we conclude that multithreading does not always guarantee a better performance over single-threading. According to our observation the *Dual-unit Threading* performs the best.

4.2.2 The Final Version

The final version of our generator is essentially an extension of the previous version. We studied several ontology schemas where we observed that a large number of concepts are connected with object properties. Therefore, in this version, we take the *object properties* into account. Typically, the object properties are inherited from the super classes. If a concept (class) is a range of an object property, subconcepts are the range of that property as well. Figure 5 shows an example of inheritance of property instances.

Unlike data properties, object properties are optional. For instance, many individuals can be found without object properties in a generated data set.

An important extension of this version is instance *materialization*, which refers to propagating instances of the concepts that are linked with other concepts in the ontology.

```
Input  : a TClass(tClass)
Output: an Individual

if tClass.getRemainingInstances() > 0 then
    tClass.decreaseRemainingInstances();
    Individual ind = new Individual(tClass.getIRI());
    foreach data_property in tClass.getSuperDataProperties() do
        ind.addRandomTypedAttribute(data_property.getPropertyName(),
        data_property.getPropertyType());
    end
    foreach data_property in tClass.getDataProperties() do
        ind.addRandomTypedAttribute(data_property.getPropertyName(),
        data_property.getPropertyType());
    end
    foreach obj_property in tClass.getObjectProperties() do
        if random() ¿ 0.40 then
            Individual linkedInd =
            CreateRecursiveLinkedInstance(obj_property.getRange());
            if linkedInd != null then
                ind.addObjectProperty(obj_property.getProperyName(),
                linkedInd.getName());
            end
        end
    end
    buffer.append(ind.getRDF());
    if buffer.size() > MAX_BUFFER_SIZE then
        oFile.append(buffer);
    end
else
    return null;
end
```

Algorithm 5: Recursive generation of linked instances

Furthermore, we implemented final version using *block structure* method. The instances are generated into blocks, which then constitute clusters of instances. We use a recursive algorithm (Algorithm 5 to generating blocks.

The instance-generation process of our generator comprises four steps.

1. The objects (`TClass`) are created in this step. An user can choose a range - *minimum* and *maximum* number – of objects.

2. In this step, the generator first instantiates the concepts which are not a range of any properties. This information is extracted by the information extractor in the previous phase. Creating such instances is expressed as shown in Algorithm 5. Algorithm 5).

Since the object properties are optional, we a set default value (40%) of probability to estimate the likelihood that a concept is linked with another concept. The generator repeats the operation recursively as long as the subsequent instances have object properties. Then, the generator writes the output into the system buffer. Once the buffer if full, the instances are moved to files.

3. In this step, the instances are generated from the concepts that are linked with others through their ranges. As in the previous step, the previous steps, the instance generation is carried out by the recursive algorithm shown as Algorithm 5).

Multithreading

In the final version, we extend the multi-threading algorithm that was developed in the initial version. Note that, since we found through our experiment with the initial version that the *Dual-unit Threading* is the most suitable approach, it is the one we use for this version. Algorithm 6 provides the pseudocode of the multi-threading approach underlying this version of the generator. The worker threads use the recursive algorithm (shown as Algorithm 5) to generate instances. However, they use two different buffers and write the output file sequentially.

5 Demonstration

We have implemented two distinct interfaces for using the generator: The Graphical User Interface (GUI) and the Command Line Interface (CLI). In this section, we demonstrate both interfaces. We explain how these interfaces assist in using the generator.

The generator is launched using the GUI by clicking on the `.exe` icon. Figure 6 shows the graphical interface.

Instructions for using to use the GUI are listed below:

1. The user chooses an ontology (until this step is done, others functionalities are disabled). Alternatively, the user can manually type the name of the ontology.
2. The user clicks on the *Load* button.
3. The generator displays information about the ontology (IRI, number of concept and properties).
4. (Optional)The user can add random properties to the concept. The user is allowed to choose any type of data properties.
5. The user saves an updated ontology either in the original file or in a new file. By default, an output file will be named *OWLInstances.rdf*, but the user is allowed to rename it.

Input : The input file ($iFile$), the output file($oFile$), the numbers of instance to generate (min and max)
Output: The number of instances generated

initialization;
$oFile$ = copyFile($iFile$); TClass[] $class\_array$ = $map\_classes$.toArray();
$worker1$ = $worker2$ = new Thread
begin /**STEP1**/
  while *(int* i = *nextClass())* $< class\_array.length()$ **do**
    $class\_array[i]$.initialization(randomInt(min, max));
  end
end
$worker1$.join();
$worker2$.join();
$worker1$ = $worker2$ = new Thread
begin /**STEP2**/
  while *(int* i = *nextClass())* $< class\_array.length()$ **do**
    if *not* $class\_array[i].isARange()$ **then**
      createRecursiveLinkedInstance($class\_array[i]$);
    end
  end
end
$worker1$.join();
$worker2$.join();
$worker1$ = $worker2$ = new Thread
begin /**STEP3**/
  while *(int* i = *nextClass())* $< class\_array.length()$ **do**
    while $class\_array[i].getRemainingInstances() > 0$ **do**
      createRecursiveLinkedInstance($class\_array[i]$);
    end
  end
end
$worker1$.join();
$worker2$.join();

Algorithm 6: Second version : with two workers

6. The user specifies the number of instances *per Classes* that should be generated. This value is specified in the *min* and *max* fields.

7. The user can choose the *materialization* option.

8. In the final step, the user clicks on the "launch generation" button. A status bar in the bottom of the interface shows the progression of the current operation.

Unlike the GUI, CLI is a command and parameter-based interface. As a result, any incorrect input will fail to launch the instance-generation process, and cause it to

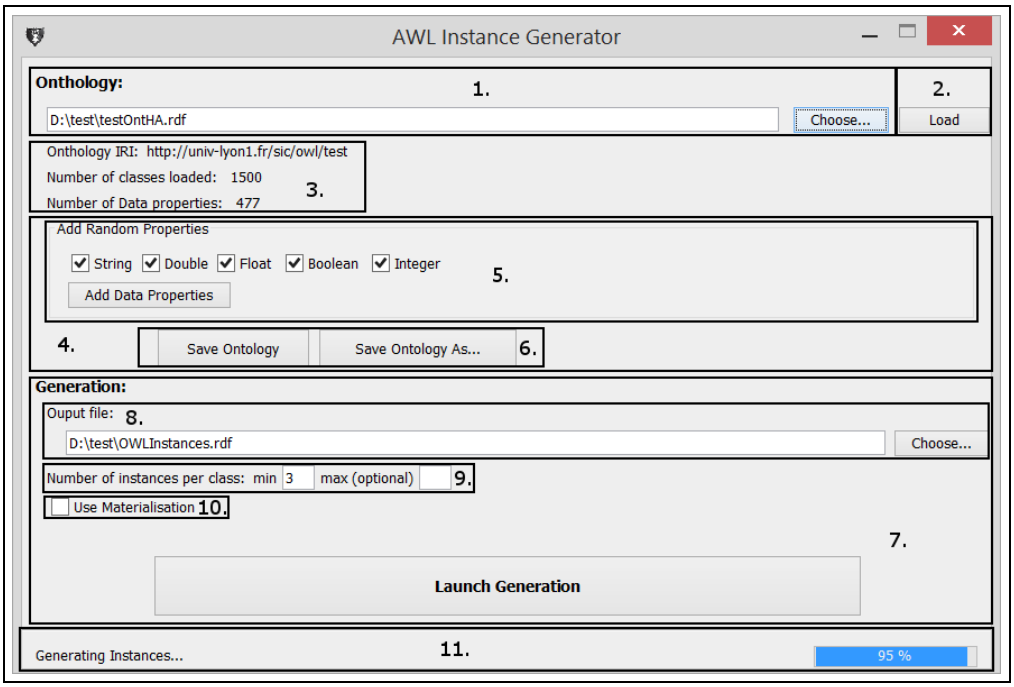


Figure 6: The Graphical User Interface

abort. For instance, if an incorrect parameter is specified, the generator is immediately aborted and an error message is issued. The command to execute the CLI for running the generator is

- `java -(assign JVM heap space) -jar ../PATH to OWL_Gerator.jar FILE -F ../PATH to Ontology Schema FILE -N Number of Instances Per Classes -O ../PATH to Output FILE`

Where:

- -F (denotes for file): path of the ontology on the file system;
- -N (denotes for number): number of instances to generate, per class (default : 3);
- -L (denotes for limit): The maximal number of instances per class (if more than N, randomly defined in the interval);
- -O (denotes for output): The output file (by default, OWLInstances.rdf in the same path than the input file);
- -M (denotes for materialization).

6 Evaluation

We conducted several experiments with GAIA. In these experiments, we evaluated two aspects: (i) the generator's the generator's capacity to read any ontology written in OWL and (ii) its performance. In this section, we describe the experiments' environment environment and the results.

6.1 Experiment Setting

We conducted experiments on the Grid5000 platform, which is a high-performance computing platform.[6] The platform is accessible through eleven different sites within and outside of France. We used the Nantes access point.[7] The site front-end has an automatic scheduler that accepts requests of reserving resources and assigns a number with the variable called 'JOB_ID'. In response to our request, the scheduler reserved a node in the cluster called *econome*. The node specification is given below:

- Processor: Intel Xeon E5-2660, 8 cores
- Memory: 64 Gigabyte
- Cache: 10 32 KB (L1) + 256 KB (L2) + 25 MB (L3)
- Network: 10 Gigabit Ethernet
- Storage: 2 TB

We transferred ontology schemas and the generator from the local machine to the remote server.

6.2 Results and Discussion

We used eight different ontologies in our experiments. We performed experiments with the widely used LUBM and our own generator. The Table shows the results of the experiment.

The table shows that GAIA successfully read and parsed all eight different ontologies whereas LUBM was able to read only the univ-bench ontology. For all but the "Univ-Bench" othology, the LUBM system failed, where GAIA succeeded

Furthermore, the LUBM generator took 110 minutes for generating 99 Gigabyte data set. The data generation rate was 900 Megabyte per minute. GAIA generated 261 Gigabyte in 48 minutes. The data generation rate 5437 Megabyte per minute, which is almost five and a half times faster than the LUBM generator. In addition, we were able to generate 1.85 Terabyte instances in 8 hours 19 minutes. This clearly demonstrates that GAIA is able to generate Big data set.

It is worth noting that the comparison shown in our experiment is biased. While generating instances, LUBM carries out a lightweight reasoning to generate instances by

[6] https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home
[7] https://www.grid5000.fr/mediawiki/index.php/Nantes:Home

Table 1: The experiments shows reading and parsing results

Ontologies	LUBM	GAIA
Univ-bench	Successfully Read and Parsed	Successfully Read and Parsed
AOO	Read and Parsed failed	Successfully Read and Parsed
AEO	Read and Parsed failed	Successfully Read and Parsed
ATO	Read and Parsed failed	Successfully Read and Parsed
APO	Read and Parsed failed	Successfully Read and Parsed
Bila	Read and Parsed failed	Successfully Read and Parsed
Biomodel	Read and Parsed failed	Successfully Read and Parsed
NCBI	Read and Parsed failed	Successfully Read and Parsed

satisfying OWL *property constraints* specifically, the `owl:intersectionOf` and `owl:someValuesFrom` constraints. The former describes an individual which is an instance of two classes. The latter describes a class of all individuals for which at least one value of the property concerned is an instance of the class description or a data value in the data range [cite]. For example, in LUBM ontology, there is a `someValuesFrom` restriction shown below:

```
<owl:Class rdf:ID="ResearchAssistant">
    <rdfs:label>university research assistant</rdfs:label>
    <rdfs:subClassOf rdf:resource="#Person"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#worksFor"/>
            <owl:someValuesFrom>
                <owl:Class rdf:about="#ResearchGroup"/>
            </owl:someValuesFrom>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

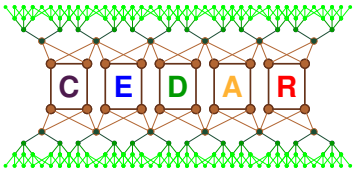
In this example, the reasoner performs reasoning to find *research assistants* who belong to the class *Persons* and *works for* at least one *research group*.

Currently, GAIA does not perform any such reasoning and therefore, the dataset produced by this generator is not complete. Additionally, according to our observation, this influences the performance as well. To be specific, LUBM needs computation time for reasoning the ontology by satisfying constraints and thus its performance (with respect to instance generation time) is lower than GAIA.

7 Conclusion

This document presented GAIA, a generic and highly efficient RDF-triple generator for large ontologies. A high-level architecture of the generator was described in this report. The implementation of the generator was detailed. The generator was evaluated by conducting several experiments. We reported the results of experiments.

During our experiment, we found that for some ontologies such as *univ-bench*, GAIA produced incorrect results if the number of instances per concept was specified more than 40000. Our future work is to improve the generator to deal with such cases efficiently.



Technical Report Number 14

GAIA

An OWL-based Generic RDF Instance Generator

Tanguy Raynaud, Rafiqul Haque, Samir Amir, Mohand-Saïd Hacid

November 2014