A Programming Interface for Creating Data According to the SPAR Ontologies and the OpenCitations Data Model

Simone Persiani[1], Marilena Daquino[2,3], and Silvio Peroni[2,3](✉)

[1] Department of Computer Science and Engineering, University of Bologna, Bologna, Italy
`simone.persiani2@studio.unibo.it`

[2] Research Centre for Open Scholarly Metadata, Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy
`{marilena.daquino2,silvio.peroni}@unibo.it`

[3] Digital Humanities Advanced Research Centre (/DH.arc), Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

Abstract. The OpenCitations Data Model (OCDM) is a data model for bibliographic metadata and citations based on the SPAR Ontologies and developed by OpenCitations to expose all the data of its collections as sets of RDF statements compliant with an ontology named OpenCitations Ontology. In this paper, we introduce *oc_ocdm*, i.e. a Python library developed for creating OCDM-compliant RDF data even if the programmer has no expertise in Semantic Web technologies. After an introduction of the library and its main characteristics, we show a number of projects within the OpenCitations infrastructure that adopt it as their building block unit.

Keywords: OpenCitations · Python · Rdf · rdflib · citation data · bibliographic metadata · Spar Ontologies

1 Introduction

Data models are crucial artifacts that datasets suppliers should make available to document data and to enable users to understand and, thus, use appropriately suppliers' data. Sometimes, data models may be created (re)using terms defined in the same ontologies with different nuances, thereby generating diversity in data representation [7]. Of course, a data model can employ clearly defined ontological terms to ensure data consistency and facilitate integration tasks.

However, even if ambiguities are entirely avoided from a terminological perspective, creating datasets compliant with a particular data model can still be a challenge for people who are not experts in the related technologies, such as OWL and RDF. Further challenges can be due to data dynamics (e.g. extensions and modifications) [22] which must be performed accordingly to the data model either to correct possible mistakes in an entity or to introduce new data. Additional complexities in data handling are introduced when the data model asks for tracking entities' provenance and changes every time an entity is modified.

To enable users (e.g. domain experts) to programmatically access the data organised according to a particular data model and to permit their modifications, applications (visual interfaces, web editors, etc.) must be developed to facilitate human-data interaction. However, an additional interface layer should be provided to permit programmers to develop such applications, since such programmers are experts in coding but not necessarily skilled in the technologies used by an underlying data model. Such an interface layer would enable creating and manipulating data transparently from the actual technologies used for their representation, such as RDF and, particularly, OWL ontologies.

The situation introduced above describes what happened in OpenCitations in the past few years. OpenCitations (http://opencitations.net) is an independent not-for-profit infrastructure organisation for open scholarship dedicated to the publication of open bibliographic and citation data by the use of Semantic Web technologies [27]. A few years ago, OpenCitations released the OpenCitations Data Model (OCDM) [7], a data model based on SPAR Ontologies [26], PROV-O [25], and other existing models, for describing all the entities in its collections, keeping track of their provenance and modifications in time. In addition of being reused by OpenCitations, the OCDM has also been recently adopted by other external projects dealing with bibliographic metadata and citations [7]. The more the OCDM is adopted, the more it is necessary to have a library to simplify the creation of applications dealing with OCDM-compliant data.

In this paper, we introduce a Python library, i.e. *oc_ocdm* [29], for enabling data owners and publishers to develop applications using OCDM-based data and provenance information. This library has already been used by OpenCitations in several components and projects, and it is the building block for all the future applications dealing with RDF data in OpenCitations' collections.

The rest of the paper is structured as follows. In Sect. 2, we introduce relevant existing libraries for simplifying the creation of RDF data compliant with data models and ontologies. In Sect. 3, we summarise the OpenCitations Data Model and list the requirements for the development of the library. In Sect. 4, we present the main characteristics of the library, including a discussion of its main modules and classes. In Sect. 5, we address the potential impact, adoption and community involved in the library development and reuse. Finally, in Sect. 6, we conclude the paper sketching out some future developments.

2 Related Works

Serving easy-to-access and effective instructions to reuse an ontology contributes to the recognition and validation of the quality of the ontology itself [13]. In recent years, several works have expanded on this aspect adapting FAIR (findable, accessible, interoperable, and reusable) principles to ontologies [12,14,31].

While ontology engineers have introduced best practices for documenting, versioning, and publishing Semantic Web artefacts, they rarely focus on the development of software to enable researchers to programmatically make use of ontologies in the early stages of their project pipelines (e.g. knowledge extraction and RDF data creation). To cope with potential data quality issues arisen by misleading interpretations of the ontologies, the development of SHACL [5] and ShEx [30] has enabled the validation of data conformance to a schema, which is defined in terms of the syntax and structure of a "shape". Nevertheless, human-readable documentation keeps being the primary way to correctly reuse ontologies, as it can more effectively convey the semantics and interpretation of ontology terms.

Only a few notable ontology providers provide effective solutions to systematically create and organise data according to an ontology. These include Python libraries, like GOATOOLS [21], which allows to reuse terms from Gene Ontology and perform data analysis, *pronto* [23], used to access specifications of the Open Biomedical Ontologies [33], or *motools* (https://github.com/motools), including a library for consuming terms of the Music Ontology. Alternatively, WYSIWYG tools, like *quickstatements* (https://quickstatements.toolforge.org) and *OntoRefine* (https://disc-semantic.uibk.ac.at/ontorefine) allow non-expert users to create and map data conforming to specific data models. It is worth noting that such efforts appear to be common in broad communities where diverse stakeholders, with more or less knowledge of Semantic Web technologies, must reuse the ontology to create data and conform to community standards. These solutions significantly prevent time-consuming quality checks, e.g. on crowdsourced data.

To the best of our knowledge, such aids are lacking in the publishing domain. Converters to transform bibliographic records into linked data according to RDA vocabularies [19] and other models exist, e.g. [9] and *bibtex2rdf* (http://www.l3s.de/~siberski/bibtex2rdf/). However, only records complying with library metadata standards are suitable for conversion and no programming interfaces are available for alternative formats, therefore excluding data produced by academic journals and venues. Similarly, well-known scholarly linked data providers [1,11,15] do not share interfaces for data creation according to their schemas [7]. In this work, we fill the gap providing a Python library for creating RDF data according to OCDM [7]. OCDM expands on several modules of the SPAR Ontologies [26], therefore allowing stakeholders in the publishing domain to easily create bibliographic and citation data regardless of their legacy formats – that could be stored according to other ontological models e.g. BIBO (https://bibliontology.com).

3 Model and Requirements

The OpenCitations Data Model (OCDM) [7] includes terms to describe bibliographic and citation data of scholarly publications. Rather than being yet another ontology, OCDM addresses a broad selection of terms belonging to the SPAR Ontologies [26], which have been conveniently collected within the

OpenCitations Ontology (OCO, https://w3id.org/oc/ontology). Such guidelines, available as open access human-readable and machine-readable documentation, are adopted by several datasets that are either created and maintained by OpenCitations or by external ontology reusers.

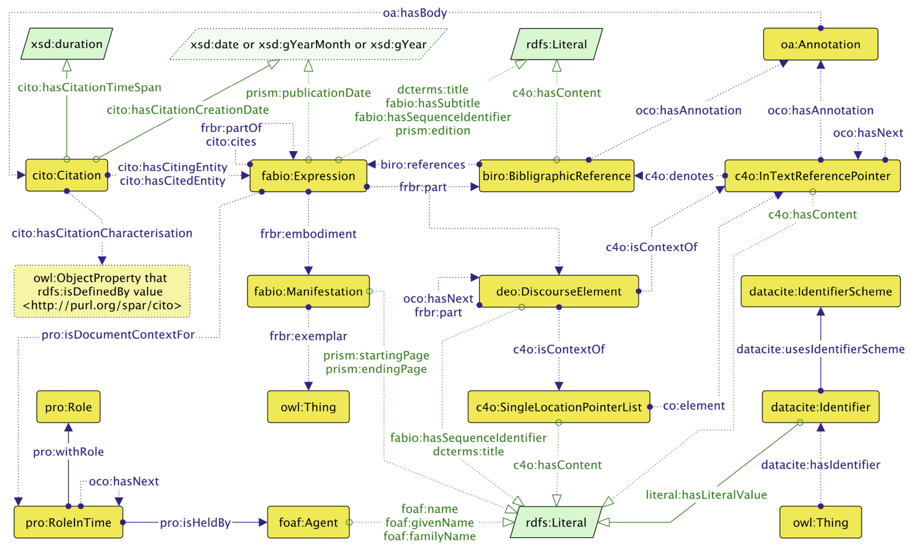

Fig. 1. The Graffoo diagram [10] of the OpenCitations Ontology. Yellow rectangles represent classes, green polygons represent datatypes, while blue and green arrows represent object properties and data properties, respectively. (Color figure online)

Specifically, the OCDM provides directives for recording dataset metadata, bibliographic entities metadata, identifiers, and provenance metadata (including versioning and provenance of changes in data). Dataset metadata include information on the distribution (e.g. a downloadable file) of the dataset. Bibliographic metadata (summarised in Fig. 1) include descriptions of bibliographic resources such as journals and articles (`fabio:Expression`), analog and digital editions of resources (`fabio:Manifestation`), in-text reference pointers (`c4o:InTextReferencePointer`), lists of pointers (`c4o:SingleLocationPointerList`), agents (`foaf:Agent`) and their roles (`pro:Role` linked to the agent via `pro:RoleInTime`), bibliographic references (`biro:BibliographicReference`), citations (`cito:Citation`), identifiers and their schemes (`datacite:Identifier` and `datacite:IdentifierScheme`).

Provenance metadata describe snapshots of data, which document the evolution of a particular entity as detailed in [28]. The provenance mechanism enforced by OpenCitations, summarised in Fig. 2, foresees an initial *creation* snapshot, potentially followed by operations like *modification*, *merge* and *deletion*, each corresponding to an additional snapshot.

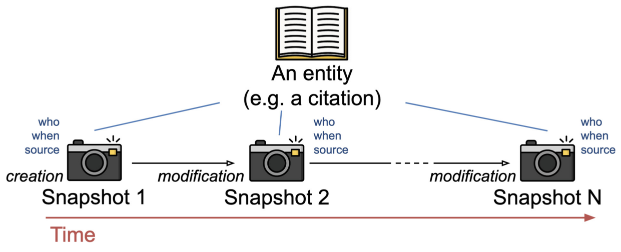

Fig. 2. The high-level description of the provenance layer of the OCDM to keep track of an entity's changes.

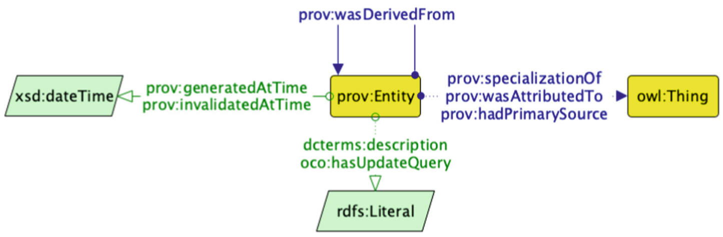

Fig. 3. The Graffoo diagram describing snapshots (`prov:Entity`) of an entity (linked via `prov:specializationOf`) and the related provenance information.

Every snapshot is linked to the described entity via `prov:specializationOf`, and to the previous snapshot via `prov:wasDerivedFrom` (see Fig. 3). Creation time (`prov:generatedAtTime`) and invalidation time (`prov:invalidatedAtTime`) of a snapshot are recorded along with the SPARQL Update query (`oco:hasUpdateQuery`) that encodes the changes applied with respect to the previous snapshot. The operation is also described with free text (`dcterms:description`), and the snapshot is linked to the source of metadata (`prov:hadPrimarySource`) and to the agent responsible for it (`prov:wasAttributedTo`).

The development of the *oc_ocdm* library was driven by the need of reengineering the existing OpenCitations' tools to make them modular. These tools should reuse basic software components, among which *oc_ocdm* has a central role. The library was developed considering the following requirements.

The first requirement was to adopt a development methodology that could make errors easier to spot at development time, thus ensuring better code quality even if with increased maintenance costs. We chose the Test Driven Development (TDD) method [2], which imposes a preliminary test design phase followed by an alternation of software development and testing.

The second requirement was the need to use a programming language compliant with the one that is used in other OpenCitations' applications, which led us to choose Python. To make the library easy to install for the final user, it was decided to package it, to manage its external dependencies and to release it on the PyPI online repository by making use of Poetry[1], a tool for dependency management and packaging in Python. In this way it is possible to manage the versioning of the library separately from that of the projects depending on it, making it simpler for users to follow the advancements in its development.

The final requirement was operational and concerned the design of a mechanism to consider the existing state of an entity defined somewhere (e.g. in a file or in a triplestore) in order to understand which modifications are applied to such an entity through the library.

4 Implementation

The Python library *oc_ocdm* (repository at https://github.com/opencitations/oc_ocdm, documentation at https://oc-ocdm.readthedocs.io/) is based on `rdflib`[2] and was developed to simplify the handling of OCDM-compliant RDF graphs, including tasks of information extraction, shape validation, editing, provenance tracking and data serialisation. It's organised as a hierarchy of subpackages, each consisting of a set of Python modules. All the main subpackages are shown in Fig. 4 and in Fig. 5, and describe the main classes they define.

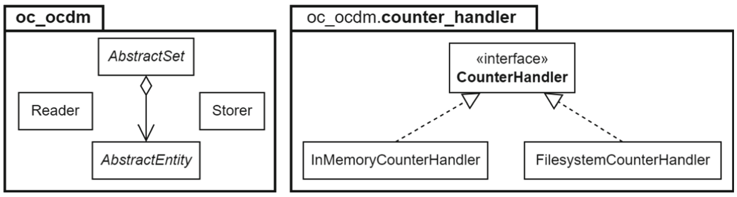

Fig. 4. The UML diagram of the main package and of the counter_handler subpackage.

The classes `Reader` and `Storer` are used for importing data from external sources and for either exporting data to a file or synchronising entities' status with an external triplestore.

Following OCDM's guidelines, all the entities are named using a URI that contains their local identifier, i.e. an incremental integer that uniquely identifies an entity among all entities of the same type. Thus, in order to enforce the uniqueness of the local identifiers for any given type of entity, the library provides a mechanism to correctly handle such counters. This functionality is provided by the abstract class `CounterHandler`, for which there currently exist only two implementations. On the one hand, the `InMemoryCounterHandler` temporarily stores counters via an in-memory data structure, with every progress being immediately lost when the instance of such class is destroyed. On the

[1] https://python-poetry.org/.
[2] https://github.com/RDFLib/rdflib.

other hand, FilesystemCounterHandler makes use of the file system to persistently read and write counters. Both CounterHandler implementations are in charge of keeping track of the last assigned integer number for each kind of entity, incrementing it by one unit when a new entity of the corresponding type is created.

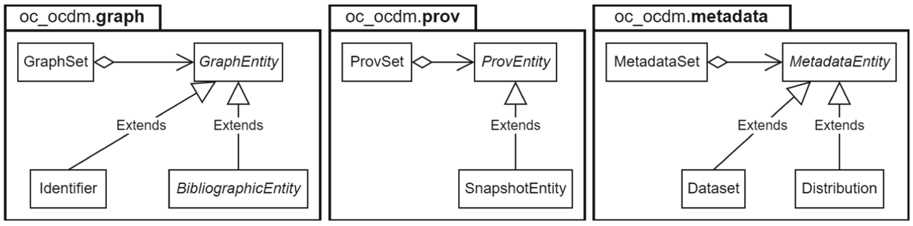

Fig. 5. The UML diagram of the graph, prov and metadata subpackages.

The **Set* classes, shown in Fig. 5, are factories defining collections of entities (**Entity* classes). Each **Set* class contains a reference to a CounterHandler instance (see Fig. 6) for managing the assignments of unique URIs to the newly generated entities. The AbstractSet class is extended by three concrete classes: GraphSet for all kinds of bibliographic entities, ProvSet for entities' provenance snapshots, and MetadataSet for metadata about the dataset and its distributions. Various subclasses of GraphEntity, ProvEntity and MetadataEntity (all subclasses of the AbstractEntity class) were defined so as to represent all the possible types of entities described in the OCDM.

In *oc_ocdm*, all the subclasses of GraphEntity and MetadataEntity are able to internally track edits. It is worth mentioning that the library enables the generation of provenance information only for non-provenance entities. In addition, each **Entity* internally holds a reference to the **Set* in which it is contained, leading to the bidirectional containment relationships shown in Fig. 6.

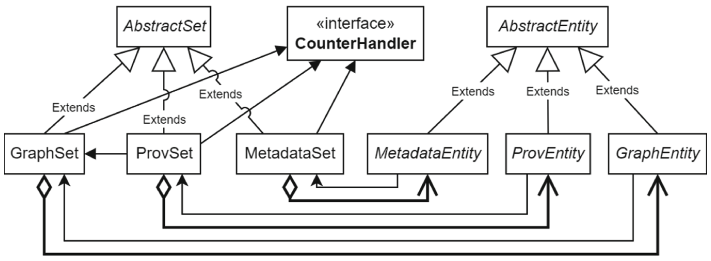

Fig. 6. A UML diagram showing the main relationships between classes in *oc_ocdm*.

4.1 Importing Data from a Persistent RDF Graph

The `Reader` class allows one to import entities from a persistent RDF graph and to parse them to produce their in-memory representations as a collection of Python objects. Entities with an `rdf:type` which is recognised by the library (i.e. the classes used the OCDM) are automatically converted into an instance of the corresponding **Entity* class and collected inside a **Set*. Additional statements about such entities that are not OCDM-compliant are, anyway, imported into the corresponding in-memory instances, even if they cannot be neither directly accessed nor modified using the methods provided by *oc_ocdm*. Every other entity is ignored and no statement about it are imported.

The `import_entities_from_graph` method processes instances of the `rdflib.Graph` class, while `import_entity_from_triplestore` sends a CONSTRUCT query to a triplestore to retrieve the statements about a single entity. Both methods enable importing only bibliographic entities (converting them into instances of a `GraphEntity` subclass) and they require an instance of the `GraphSet` class as input where to collect the imported entities.

Shape Validation. When using the methods `import_entities_from_graph` and `import_entity_from_triplestore`, the user can specify to perform shape validation on the imported graph, in order to filter out all the entities that do not respect the shape constraints described in the OCDM (constraints on a given property regarding its range datatype/class, the minimum/maximum amount of attributes associated to an entity, etc.). This operation is currently handled via the `PyShEx` library.

The shapes described in the OCDM were formalised into a proper ShExC file, that is the required input of `PyShEx`. Such a resource is included within the *oc_ocdm* package. ShEx was a design choice that we inherited from the initial phase of the development, which started a few years ago. We chose the ShExC format because of its simplicity and compactness, which makes it easy to be written and read also by non-expert users.

4.2 Data Manipulation

Oc_ocdm allows one to manipulate the content of OCDM-compliant entities only through the Python API exposed by their corresponding in-memory representations, while it does not offer a way to directly act on the persistent RDF graph. All the changes applied to a graph during a session are not made effective and persistent until the updated state of the in-memory objects is synchronised with the original dataset. The way in which the current OpenCitations' tools deal with such updates is to work with small chunks of the original dataset to reduce the impact on memory usage.

Once a **Set* instance containing the imported entities has been obtained, it is possible to access each entity individually to read its content. Several getter methods are available for each **Entity* class. For example, the title of a `BibliographicResource` instance can be obtained by calling the method `get_title` on it. For each getter method, the Python API provides also two corresponding setter methods: one for adding/modifying a value (e.g. `has_title`) and the other for removing predicate-object pairs (e.g. `remove_title`). It is worth mentioning that, with respect to methods naming, we decided to inherit the naming conventions used in the OCDM. In particular, setter methods recall the name of the ontology predicates to prevent misalignment between the OCDM and the library implementing it.

The method `remove_every_triple` modifies the **Entity* by removing every triple from its in-memory representation, without deleting its persistent counterpart. This method could be used to clear the content of an **Entity* instance to start writing on it from scratch.

In general, all the operations that can be performed on an **Entity* allow:

- the creation of a new entity;
- the modification of an entity by adding, changing, or deleting its triples;
- the merging of an entity with another one (not applicable to instances of the `ProvEntity` subclasses);
- the deletion of the entity from the graph (not applicable to instances of the `ProvEntity` subclasses).

Creation of an Entity. The creation of a new entity can be done through one of the *add_** methods made available by the `GraphSet`, `ProvSet` and `MetadataSet` classes. For example, a new `Citation` entity can be added to a `GraphSet` via the method `add_ci`. Each new instance is initialised with a triple stating its `rdf:type` and, optionally, with a user-provided `rdfs:label`.

Modification of an Entity. All the methods that apply changes to an entity (e.g. those that add or delete triples) perform preliminary checks to ensure compliance with the following constraints defined in the OCDM:

- the type of the object of a triple must comply with the one specified in the OCDM for the corresponding predicate (e.g. the value supplied to the method `has_citing_entity` must be of type `BibliographicResource`), otherwise an exception is thrown;
- predicates defined as functional properties (e.g. the method `has_name` of `ResponsibleAgent`) can be associated to at most one object. If called twice, the second value will override the first one.

Merge of Entities. The merge operation is motivated by user requirements, as it allows to manage deduplication and reconciliation of duplicated entities. Only two or more instances that belong to the same **Entity* class can be merged together. The `merge` method is used on the **Entity* to keep and specifies, as input, the other **Entity* to merge. Hence, to merge n entities together (including the one to keep), the method must be called $n-1$ times. Running the instruction `A.merge(B)` produces the following effects:

- for each predicate which is compliant with the OCDM for the particular type of the entities A and B, all corresponding objects from B are added to A (with overwriting in case of functional predicates). Every other statement from B which is not compliant with the OCDM gets ignored and it is not added to A;
- regarding the `rdf:type` predicate, the OCDM allows one to specify at most two values per entity. The first type value is mandatory and must be the same for both A and B, being in itself the requirement for enabling the merging operation. The second type value, if present in B, is added to A (or overwritten if a second value is already present in A);
- B is marked as to be deleted;
- all the other imported entities are scanned to replace B with A in all predicate-object pairs in which the object is B, thus redirecting all the references which still point to B.

Deletion of an Entity. An **Entity* can be marked as to be definitively removed from the persistent graph via the method `mark_as_to_be_deleted`. Invoking such method on an **Entity E* produces the following effects:

- E is marked as to be deleted;
- all the other imported entities are scanned to remove all predicate-object pairs having E as object, thus cleaning up dead links.

Additionally, it is possible to remove dead links also from other persistent entities that were not imported via the `remove_dead_links_from_triplestore` method from the `GraphSet` class. Such method is able to import from a triplestore all the entities that refer to at least one **Entity* which is currently marked as to be deleted and to remove the dead links from their in-memory graphs.

Marking an **Entity* as to be deleted cannot be undone. When we synchronise the deleted entity with the persistent graph, *oc_ocdm* automatically recognises that its persistent counterpart has to be completely deleted and it directly removes its persistent triples.

4.3 Change Tracking

Each **Entity* contains some private fields that are exploited by *oc_ocdm* to internally keep track of all the operations performed on it and to later reconstruct what happened. This does not apply to provenance entities, since there is no need to generate provenance information for them.

In addition to the `rdflib.Graph` that holds the current triples of the **Entity*, another `rdflib.Graph` named `preexisting_graph` is initialised during the construction of the Python instance. Such graph is intended to be read-only, as it represents the initial content of the entity at the time of importing it from the persistent graph. This allows, at any stage, to compute the changes introduced through *oc_ocdm* by making a comparison between the two in-memory graphs. There can be situations in which the initial content of an entity is unknown (e.g. when instantiating an **Entity* identified by a URI that is already assigned to an existing entity, without importing its triples from the persistent graph). In this case the `preexisting_graph` remains empty, thus the library can only interpret new triples found inside the graph as to be added to the persistent graph.

For keeping track of merging operations, each **Entity* internally sets a `was_merged` flag while also populating its `merge_list` with all the entities that were merged into it. Finally, the `to_be_deleted` flag is set when the **Entity* is marked as to be deleted.

4.4 Provenance Generation

The OCDM envisions a provenance graph composed by trees of snapshot entities that describe the evolution of bibliographic entities and their external identifiers. In *oc_ocdm*, such entities are represented with instances of the `GraphEntity` subclasses, which are the only classes having a provenance graph associated.

The most recent snapshot of either a bibliographic entity or an external identifier (e.g. DOI, ISBN, ORCID, ...) represents its current persistent content. The provenance generation algorithm provided by *oc_ocdm* is responsible for iterating the imported `GraphEntity` instances and for generating a new snapshot exclusively for those whose content has been modified through the Python API. Usually, each new snapshot invalidates the previous one, and in turn becomes the most recent snapshot for a certain entity. However, in the event that an entity is removed from the persistent graph, the corresponding snapshot to be generated must invalidate both the previous one and also itself, and no other snapshots are linked to it afterwards.

Once the user is satisfied by the changes applied to the imported `GraphEntity` instances, it is possible to automatically generate a provenance snapshot (i.e. an instance of the `SnapshotEntity` class) for each involved **Entity*. Such snapshot is intended to describe the changes produced on the related `GraphEntity` with respect to its `preexisting_graph`.

Multiple Operations on the Same Entity. The snapshot entity produced for a given `GraphEntity` can be of four different types which reflect the *oc_ocdm*'s operations, namely: creation, modification, merging and deletion. When a composition of more than one of these operations is applied to an entity, it is necessary to define a scale of priority to be followed when choosing which of them should be associated with the new snapshot. By design, the highest priority operation is deletion, followed by merging, creation and, finally, modification. For instance, if an entity is modified and later it is deleted, then the particular details of the modification are lost (i.e. they are of no interest anymore), as the operation that correctly summarises the entire process is the deletion. In such a case, a new deletion snapshot would be produced.

The Special Case of the Merging Operation. When two or more entities are merged together, a merge snapshot must be generated only for the entity that the user chooses to keep in the persistent graph (i.e. the one to which the triples of the others are added). The merge snapshot needs to keep a reference to the latest snapshots of all the entities contained in the *merge_list* (see Sect. 4.3) by means of the predicate `prov:wasDerivedFrom`. It is mandatory that the deletion snapshots for the entities in the *merge_list* are generated only afterwards, as the merge snapshot must refer to the snapshots that represent the state of the involved entities prior to their deletion. This requirement imposes a strict ordering constraint in the production of snapshots.

The Provenance Generation Algorithm. The algorithm introduced in Listing 1 describes the rationale for handling the task described in this section. Comments have been inserted in particular branches of the pseudo-code to highlight all the possible scenarios that the algorithm needs to handle. In *oc_ocdm*, it is implemented by the `generate_provenance` method of the `ProvSet` class.

The algorithm iterates (line 2) over the entities A that have been effectively merged with any number of entities B_i (i.e. $A.merge(B_i)$), that is, all those entities that have been involved in a merging while not having been consequently marked as to be deleted. For these entities, the following scenarios must be addressed:

– **scenario A.1**: A does not exist in the persistent graph, hence a creation snapshot is generated;
– **scenario A.2**: A already exists in the persistent graph but each B_i in A's `merge_list` does not exist. Since it's not possible for a merge snapshot to refer to the snapshot of any B_i as they do not exist, such a merge operation can only be interpreted by *oc_ocdm* as a modification of A;
– **scenario A.3**: A already exists in the persistent graph and at least one B_i in A's `merge_list` exists as well. In this case, a merge snapshot is generated for A that references all the latest snapshots of such B_i entities.

Then, another iteration of the algorithm is performed (line 12) on all the remaining entities E. In this case, the following scenarios can occur:

– **scenario B.1**: E does not exist in the persistent graph and was not deleted, hence a creation snapshot is generated. Had it been deleted, no snapshot would have been generated, since the deletion of a non-existing entity does not produce any change to the persistent graph;
– **scenario B.2**: E already exists in the persistent graph and it was deleted (either explicitly or as a consequence of being involved in a merging operation), hence a deletion snapshot is generated;
– **scenario B.3**: E already exists in the persistent graph, it was not deleted but it was modified. In this last case, a modification snapshot is generated.

Algorithm 1: Pseudocode for provenance generation

```
1  $resultSet \leftarrow$ an empty set
2  foreach entity such that ($wasMerged(entity)$ and not $wasDeleted(entity)$) do
3      $latestSnapshot \leftarrow retrieveLatestSnapshot(entity)$
4      if latestSnapshot is None then
           // Scenario A.1 -> Creation
5          $resultSet.add(newCreationSnapshot(...))$
6      else
7          $snapshotsList \leftarrow getSnapshotsFromMergeList(entity.merge\_list)$
8          if $wasModified(entity)$ and $len(snapshotsList) \leq 0$ then
               // Scenario A.2 -> Modification
9              $resultSet.add(newModificationSnapshot(...))$
10         else if $len(snapshotsList) > 0$ then
               // Scenario A.3 -> Merge
11             $resultSet.add(newMergeSnapshot(...))$
12 foreach remaining entity do
13     $latestSnapshot \leftarrow getLatestSnapshot(entity)$
14     if latestSnapshot is None then
15         if not $wasDeleted(entity)$ then
               // Scenario B.1 -> Creation
16             $resultSet.add(newCreationSnapshot(...))$
17     else
18         if $wasDeleted(entity)$ then
               // Scenario B.2 -> Deletion
19             $resultSet.add(newDeletionSnapshot(...))$
20         else if $wasModified(entity)$ then
               // Scenario B.3 -> Modification
21             $resultSet.add(newModificationSnapshot(...))$
22 return resultSet
```

4.5 Data Synchronisation

Generally, the last step of a workflow that involves the manipulation of an OCDM-compliant dataset consists in the synchronisation of the in-memory content of the **Entity* instances with a triplestore or a persistent RDF resource. All the relevant library operations are collected within the `Storer` class.

As far as the data serialisation task is concerned, the library supports three possible RDF file formats, namely: **N-Triples** for bibliographic entities and their external identifiers, **N-Quads** for provenance entities and **JSON-LD** (the default option for both kinds of entities).

The methods that the `Storer` class makes available enable one to work on the content of either a single **Entity* or an entire **Set* and permit considering the related export target (which can be either an RDF file or a SPARQL endpoint).

In particular, `store` and `store_all` methods are used to export respectively a single entity and an entire set of entities on the file system, while `upload` and `upload_all` are capable of generating batches of SPARQL 1.1 Update queries that are sequentially sent to a user-specified endpoint. Finally, the `upload_and_store` method combines the effects of `store_all` and `upload_all`.

Once the synchronisation task is executed, further modifications require the user to first call the `commit_changes` method either on single **Entity* instances or on an entire **Set*. Such method takes care of effectively destroying the Python objects of deleted entities and of resetting the internal state of the other ones (i.e. resetting their boolean flags and realigning the `preexisting_graph` with their updated persistent graph).

5 Potential Impact, Adoption and Community

In a previous article [7], we demonstrated the impact of OCDM with respect to a growing community, which includes a number of datasets and projects maintained by the OpenCitations infrastructure [16,18,27], a few OCDM adopters from diverse disciplines [4,20,24], a growing number of applications and services that rely on data served by OpenCitations (e.g., VOSViewer[3], CitationGecko[4], VisualBib[5], and OAHelper[6], DBLP[7] and Lens.org[8]), and data providers that align data to OpenCitations (e.g., OpenAIRE[9], MAKG, and WikiCite).

The library *oc_ocdm* has been tested and it is currently integrated into four applications and collaborative projects, namely: *Wikipedia Citations in Wikidata*[10], a project funded by the Wikimedia Foundation to extract citations from the English Wikipedia towards external bibliographic resources, transform data to RDF according to OCDM, and upload citations to Wikidata; *OpenCitations Meta*[11], a software to clean and transform tabular bibliographic metadata to RDF according to OCDM; *GraphEnricher*[12], a tool for identifiers discovery and data deduplication used to improve data quality of OpenCitations data; finally, *oc_ocdm* is used to define testing benchmarks of another Python library[13] used to perform time and provenance-aware queries on RDF datasets compliant with the OCDM.

Nevertheless, like other software developed by OpenCitations [6,8,17], also *oc_ocdm* has been developed with the aim of sharing a component that can

3 https://www.vosviewer.com/.
4 https://citationgecko.com/.
5 https://visualbib.uniud.it/en/project/.
6 https://www.otzberg.net/oahelper/.
7 https://dblp.org.
8 https://lens.org.
9 https://www.openaire.eu/.
10 https://meta.wikimedia.org/wiki/Wikicite/grant/Wikipedia_Citations_in_Wikidata.
11 https://github.com/opencitations/meta.
12 https://github.com/opencitations/oc_graphenricher.
13 https://github.com/opencitations/time-agnostic-library.

be reused in different contexts. In particular, the broader community of SPAR Ontologies adopters can benefit of this programming interface, including current adopters for (1) data creation, (2) data analysis [3], (3) ontology-based data managements systems [32] and (4) academic journals [34].

Moreover, *oc_ocdm* can be reused by scholars in the Library and Information Science domain that want to produce bibliographic and citation data according to the SPAR Ontologies, which comply with most of the requirements for bibliographic ontologies (e.g. being based on FRBR conceptual model).

6 Conclusions

In this paper, we have introduced *oc_ocdm*, a Python library for enabling the development of applications using OCDM-based data and provenance information. After showing the main requirements for the development, we have introduced its organisation in terms of Python modules and classes and we have presented its current and future uses in the context of several components and projects related to OpenCitations, being the main building block for all the applications dealing with creating and modifying RDF data in OpenCitations' collections.

In the future, we aim at continuing the development of the library adding new features and reusing other existing components. For instance, since mid-February 2021 the development of `PyShEx`, that it is used to validate input data processed by *oc_ocdm*, has been slowed down. Therefore, we plan to convert the ShExC file into a compact SHACL file and use the `pySHACL`[14] library, which is actively maintained and better optimised.

Another aspect that deserves to be properly addressed is related to the parallel use of the library. Indeed, the current release of *oc_ocdm* is designed to work correctly if no more than one instance of *oc_ocdm* needs to access the indexes used to name new entities. If this condition is not met, episodes of race conditions could easily occur with the risk of assigning the same URI to more entities, therefore compromising the consistency and validity of all the produced data.

Finally, a last aspect that deserves to be addressed concerns the possibility of detaching from the library some aspects that can be applied to any RDF dataset, and not only to OCDM-compliant data. For instance, the way proposed by the OCDM to handle provenance and, in particular, change tracking is independent from the kinds of entities to track and could be devised, in the future, as a possible plugin for `rdflib`.

Acknowledgements. This work has been funded by the project "Open Biomedical Citations in Context Corpus" (Wellcome Trust, Grant n. 214471/Z/18/Z) and the project "Wikipedia Citations in Wikidata" (Wikimedia Foundation, https://meta.wikimedia.org/wiki/Wikicite/grant/Wikipedia_Citations_in_Wikidata), and partially funded by the European Union's Horizon 2020 research and innovation program under grant agreement No 101017452 (OpenAIRE-Nexus). We would like to thank (in

[14] https://github.com/RDFLib/pySHACL.

alphabetic order) Fabio Mariani, Arcangelo Massari, and Gabriele Pisciotta for the constructive feedback.

References

1. Ammar, W., et al.: Construction of the literature graph in semantic scholar. In: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Vol. 3 (Industry Papers), pp. 84–91. Association for Computational Linguistics, New Orleans - Louisiana (2018). https://doi.org/10.18653/v1/N18-3011.https://aclanthology.org/N18-3011
2. Beck, K.: Test-Driven Development: By Example. The Addison-Wesley signature series, Addison-Wesley, Boston (2003)
3. Bertin, M., Atanassova, I., Sugimoto, C.R., Lariviere, V.: The linguistic patterns and rhetorical structure of citation context: an approach using n-grams. Scientometrics **109**(3), 1417–1434 (2016). https://doi.org/10.1007/s11192-016-2134-8
4. Colavizza, G., Romanello, M.: Citation mining of humanities journals: the progress to date and the challenges ahead. J. Eur. Periodical Stud. **4**(1), 36–53 (2019)
5. Corman, J., Reutter, J.L., Savković, O.: Semantics and validation of recursive SHACL. In: Vrandečić, D. (ed.) ISWC 2018. LNCS, vol. 11136, pp. 318–336. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00671-6_19
6. Daquino, M., Heibi, I., Peroni, S., Shotton, D.: Creating RESTful APIs over SPARQL endpoints using RAMOSE (2020). http://arxiv.org/abs/2007.16079
7. Daquino, M., et al.: The opencitations data model. In: Pan, J.Z., Tamma, V., d'Amato, C., Janowicz, K., Fu, B., Polleres, A., Seneviratne, O., Kagal, L. (eds.) ISWC 2020. LNCS, vol. 12507, pp. 447–463. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62466-8_28
8. Daquino, M., Tiddi, I., Peroni, S., Shotton, D.: Creating open citation data with BCite. In: Emerging Topics in Semantic Technologies, pp. 83–93. IOS Press, Amesterdam (2018)
9. Dunsire, G., Fritz, D., Fritz, R.: Instructions, interfaces, and interoperable data: the rimmf experience with RDA revisited. Cataloging Classif. Q. **58**(1), 44–58 (2020)
10. Falco, R., Gangemi, A., Peroni, S., Shotton, D., Vitali, F.: Modelling OWL ontologies with graffoo. In: Presutti, V., Blomqvist, E., Troncy, R., Sack, H., Papadakis, I., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8798, pp. 320–325. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11955-7_42
11. Färber, M.: The microsoft academic knowledge graph: a linked data source with 8 billion triples of scholarly data. In: Ghidini, C. (ed.) ISWC 2019. LNCS, vol. 11779, pp. 113–129. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30796-7_8
12. Franc, Y.L., Coen, G., Essen, J.P.V., Bonino, L., Lehväslaiho, H., Staiger, C.: D2.2 FAIR Semantics: First Recommendations (2020)
13. Gangemi, A., Catenacci, C., Ciaramita, M., Lehmann, J.: Modelling ontology evaluation and validation. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 140–154. Springer, Heidelberg (2006). https://doi.org/10.1007/11762256_13
14. Garijo, D., Poveda-Villalón, M.: Best Practices for Implementing FAIR Vocabularies and Ontologies on the Web. Applications and practices in ontology design, extraction, and reasoning, vol. 49, p. 39 (2020)

15. Hammond, T., Pasin, M., Theodoridis, E.: Data integration and disintegration: managing springer nature SciGraph with SHACL and OWL. In: International Semantic Web Conference (Posters, Demos & Industry Tracks) (2017)
16. Heibi, I., Peroni, S., Shotton, D.: Crowdsourcing open citations with CROCI-An analysis of the current status of open citations, and a proposal (2019). arXiv preprint arXiv:1902.02534
17. Heibi, I., Peroni, S., Shotton, D.: Enabling text search on SPARQL endpoints through OSCAR. Data Sci. **2**(1–2), 205–227 (2019)
18. Heibi, I., Peroni, S., Shotton, D.: Software review: COCI, the opencitations index of crossref open DOI-to-DOI citations. Scientometrics **121**(2), 1213–1228 (2019). https://doi.org/10.1007/s11192-019-03217-6
19. Hillmann, D., Coyle, K., Phipps, J., Dunsire, G.: RDA vocabularies: process, outcome, use. D-Lib Mag. **16**(1/2), 6 (2010)
20. Hosseini, A., Ghavimi, B., Boukhers, Z., Mayr, P.: EXCITE-A toolchain to extract, match and publish open literature references. In: 2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL), pp. 432–433. IEEE (2019)
21. Klopfenstein, D., et al.: GOATOOLS: a python library for gene ontology analyses. Sci. Rep. **8**(1), 1–17 (2018)
22. Käfer, T., Abdelrahman, A., Umbrich, J., O'Byrne, P., Hogan, A.: Observing linked data dynamics. In: Cimiano, P., Corcho, O., Presutti, V., Hollink, L., Rudolph, S. (eds.) ESWC 2013. LNCS, vol. 7882, pp. 213–227. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38288-8_15
23. Larralde, M., Philipp, A., Henrie, A., Himmelstein, D., Mitchell, S., Sakaguchi, T.: althonos/pronto: 2.4.3 (2021). https://doi.org/10.5281/zenodo.5153400
24. Lauscher, A., et al.: Linked open citation database: enabling libraries to contribute to an open and interconnected citation graph. In: Proceedings of the 18th ACM/IEEE on Joint Conference on Digital Libraries, pp. 109–118 (2018)
25. Lebo, T., Sahoo, S., McGuinness, D.: PROV-O: the PROV ontology. W3C Recommendation 30 Apr 2013 (2013). http://www.w3.org/TR/2013/REC-prov-o-20130430/
26. Peroni, S., Shotton, D.: The SPAR ontologies. In: Vrandečić, D. (ed.) ISWC 2018. LNCS, vol. 11137, pp. 119–136. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00668-6_8
27. Peroni, S., Shotton, D.: OpenCitations, an infrastructure organization for open scholarship. Quant. Sci. Stud. **1**(1), 428–444 (2020)
28. Peroni, S., Shotton, D., Vitali, F.: A document-inspired way for tracking changes of RDF data - the case of the OpenCitations Corpus. In: Hollink, L., Darányi, S., Meroño Peñuela, A., Kontopoulos, E. (eds.) Detection, Representation and Management of Concept Drift in Linked Open Data. CEUR Workshop Proceedings, vol. 1799, pp. 26–33. CEUR-WS, Aachen (2016). http://ceur-ws.org/Vol-1799/Drift-a-LOD2016_paper_4.pdf
29. Persiani, S.: opencitations/oc_ocdm (version 6.0.2) (2021). https://doi.org/10.5281/zenodo.5770647
30. Prud'hommeaux, E., Labra Gayo, J.E., Solbrig, H.: Shape expressions: an RDF validation and transformation language. In: Proceedings of the 10th International Conference on Semantic Systems, pp. 32–40. SEM 2014, Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2660517.2660523
31. Riungu-Kalliosaari, L., Hooft, R., Kuijpers, S., Parland-von Essen, J., Tana, J.: D2.4 2nd report on FAIR requirements for persistence and interoperability (2020)

32. Senderov, V., et al.: OpenBiodiv-O: ontology of the openbiodiv knowledge management system. J. Biomed. Semant. **9**(1), 1–15 (2018). https://doi.org/10.1186/s13326-017-0174-5
33. Smith, B., et al.: The OBO foundry: coordinated evolution of ontologies to support biomedical data integration. Nat. biotech. **25**(11), 1251–1255 (2007)
34. Willighagen, E.: Adoption of the citation typing ontology by the journal of cheminformatics. J. Cheminformatics **12**(1), 1–3 (2020)