

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Second Cycle Degree Programme in

DIGITAL HUMANITIES AND DIGITAL KNOWLEDGE

A methodology and a software for validating citation data and bibliographic metadata
according to the OpenCitations Data Model

Final Dissertation in

Programming and Data Structures

Supervisor Silvio Peroni
Co-supervisor Ivan Heibi

Presented by Elia Rizzetto

Session III

Academic Year
2021-2022

Table of Contents

Table of Contents	1
Index of Figures	3
Index of Tables	4
Index of Scripts	5
Abstract	6
Introduction	7
Literature review: validation languages and software	12
XML schema languages	12
Document Type Definition (DTD)	12
W3C XML Schema	15
RELAX NG	16
Schematron	17
JSON Schema	18
ShEx	19
SHACL	20
Python software for data validation	22
Jschema library	22
Python libraries for semantic validation: pyShEx and PySHACL	22
Cerberus	23
Pydantic	24
Methodology	25
Use case	25
The data	26
Intended outcome	27
Implementation methodology	27
Input	27
Validation Process	29
First level: internal syntax	31
Second level: the external syntax of identifiers	32
Third level: the existence of identifiers	33
Fourth level: semantics and compliance to OCDM	34
Output	34
Implementation: high-level overview	36
OpenCitations Data Model	36
oc_idmanager library	38

The validation software: classes and functionalities	40
Validator class	41
Wellformedness class	42
IdSyntax class	45
IdExistence class	46
Semantics class	46
Helper class	49
group_ids method	49
create_error_dict method	49
create_validation_summary method	53
Code launch	55
Implementation: technical details	59
General features	59
Identifier Managers	61
The validation software	65
Pre-process operations	66
Validating META-CSV	68
Creating and storing errors	70
Well-formedness	76
get_missing_values	78
Syntax of Identifiers	81
Existence of Identifiers	82
Semantics	83
Validating CITS-CSV	85
Software evaluation	89
Conclusions and future developments	93
Conclusions	93
Future developments	94
Bibliography	96

Index of Figures

Figure 1. Representation of the internal structure of a cell (author)	28
Figure 2. Representation of the internal structure of a cell (pub_date)	28
Figure 3. The Graffoo diagram of the OCDM.	37
Figure 4. The UML class diagram of oc-idmanager.	40
Figure 5. The required field values and the conditions of their requiredness.	45
Figure 6. A simplified UML class diagram of the validation software.	55
Figure 7. An abridged version of the taxonomy of errors.	57

Index of Tables

Table 1. Two sample table cells of META-CSV.	28
Table 2. The type-ID alignment.	48
Table 3. The imports and the dependencies of the validator.	61
Table 4. The methods of the Wellformedness class and their input data.	78
Table 5. Evaluation results (15 rows)	90
Table 6. Evaluation results (500 rows)	92

Index of Scripts

Script 1. Code Launch.	56
Script 2. The ViafManager class initializer.	62
Script 3. The is_valid method of VIAFManager.	63
Script 4. The syntax_ok method of VIAFManager.	64
Script 5. The exists method of VIAFManager.	65
Script 6. The initializer of the Validator class.	67
Script 7. The read_csv method of Validator.	67
Script 8. The process_selector method of Validator.	68
Script 9. The call of create_error_dict inside validate_meta.	69
Script 10. Reduced version of validate_meta.	71
Script 11. Sub-process of validate_meta for responsible agents.	76
Script 12. Sub-process of validate_meta for validating required field values.	78
Script 13. Validating required field values: get_missing_values.	79
Script 14. The IdSyntax class.	82
Script 15. The IdExistence class.	83
Script 16. The call of check_semantics inside validate_meta.	84
Script 17. The check_semantics method of the Semantics class.	85
Script 18. The validate_cits method.	86

Abstract

The thesis work introduces a methodology and a software for the validation of citation data and bibliographic metadata structured in tabular format, according to the OpenCitations Data Model (OCDM). In particular, the technical solution stems from the need of ensuring the validated data quality and its conformance to the data model by providing the users with machine-readable and human-readable outputs both locating possible errors in the table and providing explanations of the reasons for validation failures. The two case studies chosen for implementing the methodology concern data ingestion in OpenCitations META and CROCI databases, respectively dealing with metadata of bibliographic entities and citation data.

The validation procedure was structured on the variety of validation aspects to be handled, resulting in the following categorization: (1) validation of the table's format with respect to its own format and model specifications; (2) validation of the ID values against the syntax of the corresponding schemes; (3) verification of the IDs existence; (4) validation of semantic relations according to the OCDM, covering aspects not depending on the table format, such as the compatibility between ID and resource type for a given bibliographic entity.

After an overall introduction to the thesis work, the first chapter presents a literature review on existing schema languages and validation software. The second chapter analyzes the case study and the features of the data involved, introducing the implementation methodology. The third and the fourth chapters describe the Python software of the validator respectively at high-level and in technical detail. The fifth chapter presents an evaluation of the software based on the process execution time, and it is followed by conclusive remarks on the work including an overview of possible future developments.

Introduction

This thesis presents a methodology and a software to validate tabular documents containing citation data and bibliographic metadata according to a specific data model, i.e. the OpenCitations Data Model (OCDM) (Daquino, Peroni, and Shotton 2020; Daquino et al. 2020). The proposed methodological solution aims to ensure the quality of the validated data and its conformance to the OCDM, and at the same time help users to interpret and correct possible errors in the data they provide, returning both machine-readable and human-readable outputs that identify the location of errors in the table and explain the reasons for validation failures.

OpenCitations (Peroni and Shotton 2020) is a non-profit infrastructure organization that implements, curates, and publishes collections of scholarly citation data (which consists of links between two scholarly publications where one of the two publications is referencing the other), and bibliographic metadata (which consists of the data about a publication, such as its title, its publication date, its authors, etc.). All the OpenCitations collections are published as open data and are provided by leveraging Semantic Web technologies: everyone can freely access, query, and reuse the information stored in the databases, coherently with the general goal of the project, i.e. facilitating the communication of research endeavors and outcomes between researchers.

The main collections that have been published by OpenCitations so far include the OpenCitations Indexes, i.e. databases of citations built from bibliographic data publicly provided by controlled sources. In the OpenCitations Indexes, citations are represented as first-class data entities, meaning that each citation is treated as an entity in its own right (representing a directed link between two other entities). This allows accompanying the citation with descriptive properties, such as the citation's ID; the timespan (i.e. the temporal characteristic of the citation, consisting of the distance, expressed in time units, between the publication date of the citing entity and the publication date of the cited entity); the creation date (i.e. the date when the citation was generated by a bibliographic resource citing another bibliographic resource, with the same numerical value of the publication date of the citing bibliographic resource); and the citing and cited entity's IDs. All the publications involved in the OpenCitations Indexes, i.e. the bibliographic entities appearing as citing or cited entities for citations, are included in the OpenCitations Meta database, along with basic metadata associated with them: the identifiers, the title, the authors and editors, the publication date, the venue containing the publication (with the details about the volume, the issue, and the pages containing the publication), the type of publication (e.g. book or journal article), and the publisher. The description

of how bibliographic entities, and the relations between them, are represented in the OpenCitations infrastructure is provided by the OCDM, the OpenCitations Data Model.

The activity carried out by OpenCitations as an infrastructure organization is part of the broader context of Open Science, the movement that aims to disseminate scholarly research and make its tools (including bibliographic and citation data) and its outcomes freely accessible to everyone. In particular, OpenCitations is concerned with promoting and defending the accessibility of citation data. To achieve this goal, in 2017 OpenCitations founded, together with other partners, the Initiative for Open Citations:¹ an advocacy group formed by researchers, publishers, and other interested parties that promotes the availability of citation data that are structured (expressed in a common machine-readable format), separable (i.e. accessible without the need to access and analyze the source publications), and open (i.e. accessible and reusable for free). The availability of open scholarly citations and bibliographic metadata is crucial for the academic community and supports reproducibility and fairness in research. Nonetheless, until recently the majority of citation data was held by private companies under a proprietary license and made available only upon paying a subscription fee (Shotton 2013). In the last years, projects aimed at making scholarly citation data unrestrictedly available, such as the Initiative for Open Citations itself, have made significant progress in increasing the number of open citations, progressively reducing the gap between the number of citations that are open and the number of citations that are published behind commercial paywalls. Indeed, in 2021 the overall coverage of citation data available from open sources under a CC0 license became comparable to that from sources requiring paid subscriptions for access to data (Martín-Martín et al. 2021; Martín-Martín 2021).

Despite these encouraging results, OpenCitations aims at widening the set of available data even further, making openly available also citations that are not yet included in the existing open sources. To address this goal, in 2019 OpenCitations developed a system aimed at crowdsourcing data from identified users and created a new citation collection named CROCI, the Crowdsourced Open Citations Index (Heibi, Peroni, and Shotton 2019). The CROCI workflow allows members of the community to publish the citation data they rightfully own on the OpenCitations infrastructure. Indeed, crowdsourcing is growingly gaining relevance for the goal of enhancing the availability and accessibility of citation data, so much so that other projects have started employing and developing workflows similar to the one of CROCI for allowing users to openly publish their data (Nüst et al. 2023). Coherently, OpenCitations aims at collecting new data from users, both for citations (which

¹ Initiative for Open Citations - Homepage <https://i4oc.org/>

get ingested in the CROCI database, one of the OpenCitations Indexes) and for metadata about publications (which get ingested in OpenCitations Meta).

To enable such ingestion workflows, the data to ingest in the OpenCitations collections must be stored in tabular documents, built according to two tables: a table containing citations (CITS-CSV from now on), and a table containing metadata about publications (META-CSV from now on), both in CSV format. Both these tables, constituting the case study of this thesis, are created to contain all the information needed by OpenCitations to process the data in compliance with the OCDM. In CITS-CSV, each row represents a citation, and stores in four columns the identifiers and the publication date of the citing and the cited entity. In META-CSV, each row represents a bibliographic resource, and stores in 11 columns its basic metadata (identifiers, titles, authors, etc.). Both tables are described in two specification documents, i.e. (Massari and Heibi 2022) and (Massari 2022); these contain a number of rules about what data the document must contain and how it must be expressed, with respect to the format, the content, and the relations between the information contained in the table.

As with any other data interchange process, the ingestion of user-submitted data in CROCI and OpenCitations Meta requires making sure that the exchanged data is compatible with that required by the ingestion workflow, i.e. that it can be safely and correctly processed by the existing software of the infrastructure it is destined to. In fact, it is crucial that the data submitted by users be validated against the OCDM.

For validating CITS-CSV and META-CSV documents, it was considered the possibility of exploiting already existing tools developed for similar purposes (e.g. implementations of schema languages like JSON Schema (Pezoa et al. 2016) and format-agnostic Python softwares). However, the number and specificity of the features of the entities in the OCDM, together with the complexity of the relations between them, is hardly manageable with a general-purpose validator, making the usage of pre-built data validation software and technologies an unsuitable solution for the present use case. For this reason, this thesis proposes a new, *ad hoc* data validation software, specifically aimed at checking that the bibliographic and citation data submitted by the users comply with the data model of OpenCitations, and can therefore be processed with the software already in place within the infrastructure in order to be ingested in CROCI and OpenCitations Meta.

A core feature of the validator presented in this work, which is particularly relevant in comparing its functionalities with the ones of pre-existing softwares, is its capability to deal with possible errors in the document in a precise and case-specific manner. Indeed, a fundamental factor in designing the implementation is the intended goal of the validator: besides its more obvious purpose, i.e. to ensure that submitted data can be processed correctly and in conformance to the OCDM, the validator must

also satisfy the need to detailedly inform the users about the outcome of the validation process, in order to know which data can be successfully ingested in the dataset, and at the same time provide an information-rich explanation of the possible reasons that led certain data to fail validation.

The methodology followed in developing the validator is strongly based on the indications provided in the specification documents provided by OpenCitations (Massari and Heibi 2022; Massari 2022), which essentially describe with high specificity how users should format the tables and what data should be stored inside them. The validator, therefore, can be primarily seen as an implementation of the syntax described in the specification. However, since these documents mainly concern the well-formedness of tables, but do not cover the semantic aspects of the data contained therein, additional rules and conditions had to be verified by the validating software in order to ensure full compatibility with the OCDM. Specifically, it is required that the persistent public identifiers (like DOIs, ORCID IDs, VIAF IDs, etc.) mentioned in the tables are effectively usable as links to the real-world entity they are meant to represent, and that the relations between these IDs and the rest of the data expressed in the table fall within the ones allowed by the OCDM. In order to deal with the various natures of the aspects to validate, the implementation of the software has been guided by the following categorization:

1. Validation of the table's format (based almost entirely on the rules defined in the specification documents);
2. Validation of the ID values against the syntax (pattern) of the corresponding ID schemes;
3. Verification of the existence of the IDs;
4. Validation of semantic relations according to the OCDM, covering the aspects of data that do not depend on the format of the table (e.g. the compatibility between the IDs and the resource type assigned to the same bibliographic entity).

For the validation and verification of identifiers' syntax and existence, a software for ID-validation had already been developed by OpenCitations. However, since it managed only the set of identifiers involved in OpenCitations collections built from controlled sources (such as DOIs and ISBNs), the aforementioned software was extended to include other identifier schemes that are commonly used in the scholarly domain. In this way, it has been possible to re-use and enhance the existing software, making the process for validating the "new" identifiers more interoperable and suitable also for existing applications within the OpenCitations infrastructure.

The first chapter of this thesis presents a review of the existing schema languages and validation software. The second chapter analyzes the case study and the features of the data involved, then

describes the methodology followed to implement viable solutions with a new validator. The third chapter provides a high-level description of the software itself and its structure. The fourth chapter goes over the technical details of the implementation, analyzing the Python code. The fifth chapter presents an evaluation of the software based on the execution time of the process. Finally, conclusive remarks on the work of this thesis are provided, as well as an overview of possible future developments.

Literature review: validation languages and software

This chapter provides a brief overview of existing validation tools. The first part illustrates XML schema languages, JSON Schema, and two languages for semantic validation of RDF graphs, namely ShEX and SHACL. The second part delineates the main features of relevant Python software, dealing both with implementations of pre-existing schema languages (jsonschema, pyshex, and pyshacl libraries) and schema-independent Python libraries (cerberus and pydantic).²

XML schema languages

Document Type Definition (DTD)

The first developed means for creating XML validating schemas is the Document Type Definition (DTD). Originally included in SGML (Standard Generalized Markup Language), from which XML derives, DTDs consist of an extended context-free grammar, expressed in a notation that is similar to Extended Backus-Naur form (EBNF) (Kilpeläinen and Wood 1997; 2001).

XML DTD (henceforth just “DTD”), essentially a subset of SGML DTD (Lee and Chu 2000, 1), is defined inside the W3C recommendation for XML 1.0, as part of the language itself (Bray et al. 2008).

DTDs are a way to describe XML documents; as such, they can be used as validation schemas by a parser: many XML processors are able to validate a document against the schema defined by its DTD, since, being the first W3C tool for XML validation, it has rapidly become very popular. In order to be validated, an XML document must have a corresponding DTD specified in the Document Type Declaration by being either defined locally or stored at a URL referenced in the Document Type Declaration.

For the purpose of validation, the DTD mainly defines:

- which *elements* the document must or can contain, their structure, their quantity, and their order (any element in the document that is not defined in the DTD is not a valid element);

² A more complete investigation of the general aspects of data validation would certainly present and compare other schema languages and software, too. For the purpose of this work, though, we focus on the main XML and JSON schema languages, in the hope that they meet the need to gain a deeper understanding of the main concepts related to the validation of documents for data interchange.

- the *attributes* for the elements, whether they are required or not, their “datatypes” (among the basic XML built-in types, such as ‘CDATA’, ‘ID’, etc.), and their default or fixed values.

DTDs define the XML document by means of element and attribute list declarations. An element declaration defines the content model for an element, i.e. a regular expression representing the allowed internal structure for that element. A content model can describe an element as containing text only (#PCDATA keyword, i.e. “parsed character data”), element content (the keywords are the non-terminal symbols representing other elements defined elsewhere within the DTD), mixed content (#PCDATA and other elements), any type of content (ANY keyword) all of which must be defined in the DTD in order for the document to be valid, or no content at all (EMPTY keyword). When a content model specifies more than one non-terminal symbol, the keywords can be concatenated with a comma (,) or unioned with a vertical bar (|), and grouped with parentheses (()). This is particularly useful since quantifiers (?, *, and +) can be used to specify whether we expect, respectively, *0 or 1*, *0 or many*, or *1 or more* instances of the element (or group of elements) preceding the quantifier.

For example, the following declarations specify that the element `person` must contain, in this precise order, the elements `name` and `age`; the element `gender` is not mandatory, but if present it must be nested inside `person`; the last element must be either `company` or `university`; the children `age`, `gender`, `company`, and `university` contain raw text (#PCDATA); the element `name` can either contain `first` followed by `second`, or only `first`.³

```
<!ELEMENT person (name, age, gender?, (company | university))>
<!ELEMENT name (first_name, second_name?)>
<!ELEMENT age #PCDATA>
<!ELEMENT gender #PCDATA>
<!ELEMENT company #PCDATA>
<!ELEMENT university #PCDATA>
<!ELEMENT first_name #PCDATA>
<!ELEMENT second_name #PCDATA>
```

Attribute lists are declarations defining what attributes an element can take, what XML datatype they have, whether they are required, and their default or fixed values. They have the form `<!ATTLIST`

³ It is worth noticing that the content model must always be deterministic. For example, if we wanted to declare the structure of the `name` element, we might want to use the following non-deterministic expression: `<!ELEMENT name (first_name | (first_name, second_name))>`. By requirement, this is not accepted as a valid content model, since the validating XML processor, being at the `first_name` node in the XML tree, would not be able to know which `first_name` in the model is being matched, unless it looked ahead to determine whether there is a following `second_name` node (Bray et al. 2008, app. E: Deterministic content models). The requirement of deterministic content models, inherited from SGML, can become a problem, especially when it is impossible to convert a non-deterministic expression into a deterministic one (Vlist 2002, sec. 7.4.1.3; Walmsley 2012, 279–80).

`elementName attributeName TYPE DEFAULT>`, where `TYPE` is one of the built-in datatypes of XML and `DEFAULT` specifies the default literal value of the attribute or whether a value is required (`#REQUIRED` keyword), can be omitted (`#IMPLIED` keyword), or is fixed (`#FIXED` keyword; in this latter case, also a default value can be specified).

Considering the previous example, in case we want to specify that an element `person` must have an attribute which defines a unique identifier within the document (`ID` XML datatype) and the element `university` can have a “European” attribute with a value matching either “Yes” or “No”, with a default value of “Yes”; then the previous example should be extended as follow:

```
<!ATTLIST person uid ID #REQUIRED>
```

```
<!ATTLIST university European (Yes | No)4 #IMPLIED>
```

Despite its popularity, due to the fact that it has been built into XML and it provides, as a metalanguage for describing markup documents, an efficient way of defining the basic structural features of an XML instance, DTD has some major drawbacks:

- it does not use XML syntax
- it does not support XML namespaces (which allow importing vocabularies from external sources), since the birth of DTD predates their definition by W3C; thus, all the elements of the XML file must be declared in the DTD
- datatypes can be constrained only for attributes, and very weakly (only with some XML types)
- it does not provide the possibility to constrain the number of occurrences of elements
- it makes it hard, due to its syntax, to define unordered sequences of elements in the content model

⁴ This datatype is the Enumeration datatype, representing a group of possible literal values for the specified attribute. If we want the attribute to be a literal but do not need to specify a set of admitted values, we just specify the `CDATA` datatype instead.

W3C XML Schema

In order to provide solutions to some of the flaws of DTD, the W3C has come up with a new, grammar-based schema language for providing a model of XML instances.

W3C XML Schema (often abbreviated as XSD, standing for XML Schema Definition), was published in its first version as a W3C standard recommendation in 2001; XML Schema 1.1, introducing new features, was made an official recommendation in 2012.⁵

The first thing to notice about XSD is that schemas are written in XML syntax, making it easier for users to define a schema for their XML document, as they do not need to learn a different notation, as well as providing the possibility of manipulating the schema directly with XML processors (e.g. to check the schema's well-formedness or to query it).

Moreover, it supports namespaces (introduced in XML in 1999), allowing for the use of multiple XML vocabularies in the schema definition and making it possible to avoid conflicts in the use and interpretation of elements and attributes with the same name, but different purposes.

The most impactful innovation with respect to DTD, though, is probably the possibility to define a wider set of datatypes in the validation schema, for both elements and attributes.

In XSD, there are two categories of data types: simple types and complex types.

Simple types, which can be associated to both elements and attributes, are either primitive, atomic types of predefined meaning (like boolean, string, float, date, anyURI, etc.) or derived simple types, i.e. user-defined types built from primitive types.⁶

Complex types are intended to describe the text in the XML document that can contain markup, therefore they can be used only for elements, since attributes do not have children elements nor other attributes. By creating a complex type, one can define the structure of the document with a high level of granularity (e.g. contrary to DTD, it is possible to define precisely the minimum and maximum

⁵ The latest edition of XML Schema 1.1 is described via two different specifications: XML Schema Part 1: Structures (Mendelsohn et al. 2004) and XML Schema Part 2 (Biron and Malhotra 2004): Datatypes. W3C provided also an introductory non-standard document explaining the language in a more usage-oriented way, the XML Schema Part 0: Primer (Fallside and Walmsley 2004).

⁶ This can be done by means of restrictions, lists, and unions. Restriction types are declarations of further constraints to an atomic type, called facets, which can specify a range of values, a regular expression to be matched, and the like. List types are sequences of whitespace-separated atomic values, each of which can optionally be a user-defined type by restriction or union. Union types are sets defining the simple types to which a value can conform (Walmsley 2012, 128–55).

number of children for any given element, or declare, though with lesser precision, whether the elements can be unordered).

Another feature introduced with XSD (in version 1.1) is the possibility of validating data against rules, called assertions, based on XPath 2.0 (Robie et al. 2010) expressions. Assertions are defined on types; for any given element or attribute in the XML document, if the type declared for that element contains an assertion, the XPath expression of the assertion must evaluate to true for the document to be valid. Although, for a node of a given type, XPath expressions can access only elements and attributes within the scope of that node or within its descendants (not being able to traverse the tree “upwards”), assertions provide great flexibility to the validation process, being particularly useful when the value of an element or attribute depends conditionally on the value of other components.

RELAX NG

RELAX NG (Regular language for XML New Generation) (Clark and Murata 2001), was developed within an OASIS Technical Committee⁷ in the same years as XSD, being published as a Committee Specification in 2001.

RELAX NG has great expressive power. It includes many of the features of XSD, such as the support of namespaces and datatypes (although they need to be imported in the schema from an external schema), but has a simpler syntax, which can be written both in XML and a compact DTD-like notation. Moreover, it admits non-deterministic expressions, which are not possible in DTD nor in XSD, and it allows to treat attributes in the same way as elements in the definition of the content model.

Compared to XSD, RELAX NG also have some disadvantages, like the following (Walmsley 2012, 13):

- it does not support assertion with XPath expression;
- it does not provide the possibility of defining derived datatypes, as XSD does;
- it is intended only for validation, therefore it is not able to provide information to the XML processor on how to process the instance;
- it does not allow to specify identity constraints, i.e. uniquely identify nodes in XML document and verify the integrity of internal references.

⁷ https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=relax-ng

Schematron

Schematron was created by Rick Jelliffe in 1999; it has later been standardized by the International Organization for Standardization starting from 2006, leading to the publication of ISO Schematron in *ISO/IEC 19757-3*,⁸ whose latest version dates back to 2020.

Unlike the XML schema languages we have touched upon so far, which are grammar-based schema languages, Schematron is rule-based. This means that instead of being prescriptive, and trying to explicitly describe a document (for example defining the content model for an element), it checks the validity of a document by applying boolean tests (Ray and Maden 2003, sec. 4.4.5). Due to its fine granularity, Schematron is particularly suitable for checking what is often called “business rules”, i.e. relationships between the data within the XML document (or between internal and external data), such as co-occurrence constraints, conditional requirements, etc. On the other hand, it results in lengthier schema instances than grammar-based languages when it comes to validating the document structure and syntax; in fact, as noted by its creator,⁹ it is better used in conjunction with languages like DTD or XSD.

Schematron is in XML syntax. Its key parts are patterns, rules, assertions, and reports. Patterns are the basic building blocks of a Schematron schema, to be intended as a set of rules grouped together on the basis of some arbitrary criterion (they do not refer to the structure of the document). Each `<pattern>` element contains `<rule>` elements. Rules have a `context` attribute that takes as value a query (typically expressed in XPath): the components of the document that match the query are the ones to be checked for the rule. Each rule contains `<assertions>` and `<report>` elements and both of these have a `test` attribute. The value of the attribute is an XPath boolean expression: assertions fail when it evaluates to false, while reports succeed when it evaluates to true. Notably, `<assertions>` and `<report>` elements also have text content, in natural language, addressed to the user: it should be a human-readable explanation of an error (assertion) or contain any relevant information about the context (report); schema implementers not only have full control over the user messages, but are *required* to do so, since Schematron does not have built-in messages of its own.

Overall, Schematron’s syntax is extremely simple, mostly because the great majority of the operations are dealt with XPath (querying and boolean expression). One important feature of this schema language is the focus on providing an information-rich output, both machine-readable (it is in XML, therefore it, too, can be queried, processed, and validated, just like any other XML document) and

⁸ ISO/IEC 19757-3, Third edition, 2020-06: Information technology - Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation using Schematron <https://www.iso.org/standard/74515.html>.

⁹ Overview of Schematron by Rick Jelliffe <https://www.schematron.com/home/overview.html>.

human-readable. In fact, it aims at providing a fine-grained measure of the validity of a document, shifting from the concept of validity as a binary condition, to the representation of its “semantic” aspects: not only “is the document valid?”, but also “how is it invalid, and why?” It is left to the schema implementer, of course, to specify this information, making use of customizable user messages.¹⁰

JSON Schema

As JSON kept growing in popularity as an alternative to XML for data interchange, many of the tools and technologies revolving around XML, such as its schema languages, started being developed for JSON too.

The most popular mainstream schema language for validating JSON instances is JSON Schema. It was first presented as a draft in 2013 and has been enhanced ever since; the most recent version of the specification, *Draft 2020-12*,¹¹ is published in two documents (Wright et al. 2022; Wright, Andrews, and Hutton 2022), which are currently IETF¹² Internet Drafts (I-Ds).¹³ So, despite its wide number of implementations in various languages and for various purposes,¹⁴ JSON Schema has not yet reached the status of standard, mainly due to the fact that its specification falls short of providing a formal definition of the language, that in turn has been attempted by scholars external to the organization (Pezoa et al. 2016).

JSON Schema is written in JSON, therefore any JSON Schema instance must also be a valid JSON document. The minimum valid schema is either an empty JSON object `{ }` or `True`, both of which would match any instance document; JSON Schema, in fact, has a permissive, constraint-based “behavior”: by default, anything that is not constrained is allowed.

The fundamental keyword for defining a schema (i.e. for restricting the minimal valid schema) is the `type` keyword, which of course specifies the schema’s data type; it takes as value a string (or an array of strings) representing JSON types (`string`, `number`, `object`, `array`, `boolean` and `null`), plus the `integer` type.

Each type has its own keywords that can be applied to it, e.g. `string` can be constrained with keywords defining maximum or minimum length, a regular pattern to be matched, and even built-in

¹⁰ https://www.schematron.com/home/whyisschematrondifferent_.html

¹¹ <https://json-schema.org/specification.html>

¹² Internet Engineering Task Force <https://www.ietf.org/>

¹³ <https://www.ietf.org/how/ids/>

¹⁴ <https://json-schema.org/implementations.html>

formats (email, date-time, duration, etc.). The most important type is the `object`, since it is itself a valid schema; we can define and constrain the admitted values for JSON properties (key-value pairs) of an object of the instance document via the `properties` keyword. JSON Schema allows doing so with great flexibility: while the default “behavior” of the language is permissive, meaning that anything that is not constrained is allowed, we can easily specify whether additional properties are tolerated, whether undefined additional properties should conform to certain subschemas, whether and which properties are required, etc.

Schemas can be combined together by means of keywords – `allOf`, `anyOf`, `oneOf`, and `not`, which take as values an array of (sub)schemas – representing boolean operators AND, OR, XOR, and NOT.

A very useful feature of JSON Schema is that it enables to require properties and apply schemas conditionally, i.e. depending on the presence of other properties (`dependentRequired` and `dependentSchemas` keywords) and on the outcome of the application of other schemas (in the form of if-then-else chains). This feature, together with the possibility of using keywords equivalent to boolean operators, adds to JSON Schema possibilities that are typical of rule-based schema languages. In a way, we can say that JSON Schema comprises characteristics of both rule-based and grammar-based schema languages.

On top of all this, JSON Schema provides the possibility to add human-readable documentation to the schema via specific keywords.

ShEx

Shape Expressions (ShEx) was first presented in a paper by Prud'hommeaux, Labra Gayo and Solbrig (2014); the latest version, 2.1, is published as a W3C Community Group Report (Prud'hommeaux et al. 2019).

As written by its creators (Prud'hommeaux, Labra Gayo, and Solbrig 2014, 1):

“The Shape Expressions language (ShEx) is intended to perform the same function for RDF graphs as Schema languages to XML. It can be used to validate documents, communicate expected graph patterns for interfaces, and generate user interface forms and code. The syntax and semantics of Shape Expressions are designed to be familiar to users of regular expressions (specially RelaxNG). The conspicuous difference is that RDF data is a set (of triples) while regular expression data is a sequence (of characters). Regular expressions correlate an ordered pattern of atomic characters and logical operators against an ordered sequence of characters.

Shape Expressions correlate an ordered pattern of pairs of predicate and object classes and logical operators against an unordered set of arcs in a graph.”

ShEx schemas can be written in three (isomorphic) serialization formats: a JSON-LD syntax (ShExJ), an RDF representation (ShExR), and a compact, more human-friendly notation (ShExC), similar to Turtle or SPARQL.

A ShEx schema is a set of *shape expressions* that constitute a description of an RDF graph. Each shape expression comprises a logical combination¹⁵ of *node constraints* and *shapes*.

Node constraints specify the characteristics of a given node in the data graph (the *focus* node) without regard to its neighborhood, i.e. the set of triples that have that node as a subject (outgoing arcs) or object (incoming arcs). Node constraints are meant to define the kind of node (literal, blank node or IRI), its (XSD) datatype, or a set of possible values.

Triple constraints, on the other hand, define the triples that are in the neighborhood of the focus node, specifying the predicates, directionality, cardinality (i.e. the required number of matching triples), and values.

Shape expressions are associated with the target node they refer to in the data graph by means of a *shape map* external to the schema.

The output of the validation process is another shape map that only says which nodes validate and which do not, but carries no information about the error.

SHACL

Shapes Constraints Language (SHACL) (Kontokostas and Knublauch 2017) was first designed in 2015 and since 2017 has been a W3C Recommendation. It is divided into two parts: SHACL Core and SHACL-SPARQL: the former defines the core vocabulary to define shapes and basic constraints, while the latter describes a SPARQL-based extension to define more complex constraints using SPARQL.

SHACL employs pure RDF syntax, therefore any RDF serialization format – such as Turtle, JSON-LD or RDF/XML – can be used to define shapes; these are defined in a shapes graph that is effectively

¹⁵ Within a shape expression, shapes and node constraints can be combined via logical operators AND, OR or negated with NOT.

an RDF graph; the shapes graph therefore can either be in a separate document or integrated inside the data graph to validate.

In SHACL, there are two kinds of shapes: node shapes and property shapes.

Node shapes declare constraints on a node (the *focus* node¹⁶), while property shapes declare constraints on the values that can be reached from a focus node by the specified path. The values of the property specifying a path of a property shape can be IRIs or SHACL property paths. SHACL property paths are a subset of SPARQL 1.1 property paths encoded in RDF (Labra Gayo et al. 2017, vol. 7, sec. 7.7.5.6).

To declare constraints, SHACL associates to each shape one or more constraint components; these can be either built-in, i.e. core constraint components, or SPARQL-based components defined by the user. Following Pareti and Konstantinidis (Pareti and Konstantinidis 2021, 6), we can classify constraint components into three main categories: *graph-structure* components, defining constraints that are to be evaluated at the level of triples in the graph; *filter* components, restricting the values of nodes independently of the triples in the data graph; and *logical* components, defining the logical operators (AND, OR, XOR and NOT) over the rest of the constraints.

A shape and the set of nodes in the data graph to be validated against that shape are associated to each other via a target declaration. Target declarations are of four kinds: *node* target declarations specify that the constraints are applied to the one specified node; *class-based* target declarations define as target nodes of the constraint all the nodes that are instances of the specified class;¹⁷ *subjects-of* target declaration and *object-of* target declaration associates the shape to all the nodes that are respectively subject and object of the specified property in the data graph.

As for the output of the validation, it is worth noting that the SHACL specification defines a validation report, too. If the data graph is not valid, the validation output will be an RDF graph representing validation errors and the relative metadata, such as the focus nodes that failed the validation, the value, a human-readable message, etc.

¹⁶ “An RDF term that is validated against a shape using the triples from a data graph is called a focus node” (Kontokostas and Knublauch 2017).

¹⁷ “A node X is a SHACL instance of a class C if X `rdf:type/rdfs:subClassOf`* C” (Labra Gayo et al. 2017, vol. 7, para. 5.7.2).

Python software for data validation

Considering the fact that the software developed by OpenCitations, especially for what concerns the OpenCitations Indexes, is written using the Python programming language, the third section presents python-based software for data validation. The adoption of Python-based solutions fosters the interoperability of the newly developed software into the OpenCitations system.

Jschema library

Jschema¹⁸ Python library is one of the several implementations of JSON Schema. It was created by Julian Berman in 2012 and its last version was released in November 2022. It allows to validate JSON documents in Python by defining a schema with JSON Schema notation (with full support up to Draft 7). Both the document instance and the schema must be represented as Python dictionaries.¹⁹

The simplest form of validation with jschema takes place by invoking the `validate` function, which takes as input the document dictionary and the schema dictionary and returns a boolean; as soon as an error is found, a `ValidationError` is raised.

To get more control over the validation process and output, we need to instantiate a `Validator` class object and pass to it the defined schema; by doing so, it is possible to call a method (`Validator.iter_errors`) which lazily yields *all* the errors in the document, each of which is represented as a `ValidationError` object, and returns an iterator containing them. From here we can access information about each error, such as the path to the error location in the document, the path to the critical location in the schema, a user message explaining the error, etc.

On top of that, jschema provides out-of-the-box functionalities to query the errors by providing parameters like the error type or a given location of the instance document.

Python libraries for semantic validation: pyShEx and PySHACL

Two Python libraries have been developed for validating RDF graphs with the languages mentioned above: pyShEx, an implementation of SHEX, and pySHACL, and implementation of SHACL.

¹⁸ <https://github.com/python-jschema/jschema>

¹⁹ In this regard, since the document to validate is always processed as a dictionary, it is possible to validate documents in other data serialization formats, like YAML or TOML, as long as they can be converted into Python dictionaries.

PyShEx²⁰ is a library created by Harold Solbrig; it is currently integrated into OpenCitation's Python library `oc_ocdm` to validate user input data. PySHACL²¹ is a library created by the RDFLib community.

Both packages rely on another Python package, RDFLib,²² to manage RDF graphs. The main differences are due to the different potential of the languages they implement: PyShex allows for other functionalities besides validation and can leverage the use of compact syntax ShExC for the definition of the schemas.

PySHACL is in general better maintained and optimized than PyShEx; for this reason, Persiani et al. (Persiani, Daquino, and Peroni 2022) have suggested to re-implement the semantics data validation functionalities in `oc_ocdm` with `pySHACL` instead.

Cerberus

Cerberus²³ is a dependency-free, lightweight, Python library created for data validation by Nicola Iarocci in 2012; its latest release was in May 2015 and it is currently maintained by the author himself and the community.

Validation with Cerberus is based on the definition of a schema that takes the form of a mapping (typically a dictionary); the schema's values specify the validation rules that the corresponding field should conform to. Cerberus built-in validation rules include a wide range of possibilities, most of which are given in `jsonschema` and other validation libraries too: assigning a datatype to a field (either vanilla Python datatypes or types defined by the user), restricting the set of possible values and assigning default ones, declaring the dependency of a field on the presence of others, defining regular expressions patterns, etc.

In addition, it is possible to define very simple custom validation rules and integrate them into the schema.

The schema is passed into an instance of the `Validator` class, assuming its functionalities; validation is performed simply by calling the `validate` method, with the mapping representing the document as an argument.

Each single error, if any is found, is represented as a `ValidationError` object, convertible in a Python dictionary, which is very rich in information, including the exact location of the error in the validated document, its type, a human-readable explanation, etc.

²⁰ <https://github.com/hsolbrig/PyShEx>

²¹ <https://github.com/RDFLib/pySHACL>

²² <https://github.com/RDFLib/rdfLib>

²³ <https://docs.python-cerberus.org/en/stable/>

Pydantic

Pydantic²⁴ is a Python library created and maintained by Samuel Colvin; its first release dates back to 2017 and the latest version (v.10.1.4) to December 2022. Since then it has been constantly revised and updated and has progressively become one of the most used pieces of software for validation in Python.²⁵

The library utilizes a class-based approach, where data models are defined as classes using canonical Python, eliminating the need for a separate schema definition language. The basic principle of the library is to define classes that inherit from pydantic's `BaseModel` class, representing the expected structure of the data to be validated. These classes include attributes that define the required fields in the object to be validated, and can also specify optional fields. The class attributes are annotated using Python's type hints,²⁶ allowing for the use of built-in types as well as custom objects. Additionally, pydantic provides type-specific functions that can be used to derive new types by restricting built-in data types, such as setting the minimum and maximum length of a string or using regular expressions. Custom validation rules can be defined by adding the `@validator` decorator before defining a validation function inside the model class.

When an instance of data, often represented as a Python object or dictionary, is passed to a `BaseModel` class, pydantic parses and validates it against the defined model. In case the instance does not conform with the model, a `ValidationError` is raised, which contains information about the type of error, its location within the input data structure, and a default human-readable message. These `ValidationError` objects can then be obtained and processed using try-catch blocks.

One of the key advantages of using pydantic is that the parsed data structures will be instances of the model class, with all its functionalities. This includes the ability to export the model to JSON Schema, which facilitates the interoperability of data between different systems.

Due to its high flexibility and the number and range of functionalities, its ability to deal easily with complex structures, and the fact that it is extensible as well as fast,²⁷ pydantic is widely adopted as a validation tool by a vast and increasing number of private users and companies.

²⁴ <https://docs.pydantic.dev/>

²⁵ In January 2023 it has totalled more than 42 millions downloads from PyPI repository. See <https://pypistats.org/packages/pydantic>

²⁶ <https://docs.python.org/3/library/typing.html>

²⁷ Pydantic's creator Samuel Colvin states in the website that "most of the library is compiled with Cython giving a ~50% speedup, it's generally as fast or faster than most similar libraries". <https://docs.pydantic.dev/#rationale>.

Methodology

This chapter describes the methodology followed in implementing the validation software. First, an overview of the specific use case is provided, along with a brief synopsis of the sources defining the model to which the validated data should conform. Then, the intended outcome of the implementation is presented. Lastly, a sub-section explains the methodological approaches adopted, starting from the relevant features of the input, then touching upon the general criteria on which the process is based, and finally presenting an overview of the output.

Use case

As mentioned in the introduction, the purpose of the implemented software is to validate citation data and bibliographic metadata submitted by users, in order for it to be later ingested in the corresponding indexes in the OpenCitations infrastructure, respectively CROCI and OpenCitations Meta. CROCI (Crowdsourced Open Citations Index) (Heibi, Peroni, and Shotton 2019a) was created to involve the community of scholars, publishers, and institutions in the process of making citation data open²⁸ by means of submitting it to OpenCitations. OpenCitations Meta,²⁹ released in December 2022, is the database collecting the metadata for all the publications involved in any of the OpenCitations Indexes³⁰ (COCI, the index of citations from Crossref (Heibi, Peroni, and Shotton 2019b);³¹ DOCI, the index of citations from DataCite;³² POI, the index of citations from PubMed;³³ and CROCI).

The ingestion of crowdsourced data requires a validation process. This is intended to ensure the quality of the submitted data and the possibility to process it with the existing software without loss of information.

Validating data means checking the conformance of the data itself to a given model, schema, or set of rules. For the use case presented here, the definition of how valid documents should be formed follows the instructions provided by Massari and Heibi (2022) and Massari (2022a) according to the OpenCitations Data Model (OCDM) (Daquino, Peroni, and Shotton 2020; Daquino et al. 2020), and

²⁸ “A bibliographic citation is an open citation when the data needed to define the citation are freely available, downloadable and reusable” (Peroni and Shotton 2018a).

²⁹ <https://opencitations.net/meta>

³⁰ <https://opencitations.net/datasets>

³¹ <https://www.crossref.org/>

³² <https://datacite.org/>

³³ <https://pubmed.ncbi.nlm.nih.gov/>

the suggestions, integrations, and updates collected during in-person meetings with the research team of OpenCitations.

The data

The data to validate is contained in tabular documents of two types (Massari and Heibi 2022): META-CSV and CITS-CSV. CITS-CSV is a table containing citations, where each row represents one citation, and the four columns store the values for the identifiers and the publication date of the citing and the cited bibliographic resource. Each citation, i.e. the content of a row, represents an entity in its own right, and constitutes a piece of data to be included (if valid), in CROCI, the Crowdsourced Open Citations Index.

META-CSV, on the other hand, contains bibliographic metadata about documents. Each row represents a bibliographic resource, and the eleven columns specify: the identifiers associated with the resource; the title; the surname, name, and identifiers of its authors and editors; the publication date; the venue (i.e. another bibliographic resource containing the represented document, e.g. the journal containing the article represented in the row); the volume of the venue containing the document; the issue of the venue containing the document; the page range; the type of publication; and the name and identifiers of the publisher. The data contained in a valid META-CSV will be ingested in OpenCitations Meta.

For both CITS-CSV and META-CSV table instances, the maximal “unit”, i.e. entity, eligible for ingestion is the row: if a row contains one or more errors in any of its fields, the whole row, with all its fields, is discarded and marked as invalid; at the same time, a row that does not contain any error is accepted, regardless of the validity of the other rows in the same table.

The rules for producing CITS-CSV and META-CSV tables include the specification of the datatypes (date, identifier, simple string), the format of the field values (e.g. which separator should be there in case of multiple items in a given field), the (conditionally) required fields, and the like.

In addition to what is specified in the tables specification (i.e. Massari and Heibi (2022) and Massari (2022)), it is important to consider other aspects, that are relative to other criteria, i.e. rules which are not related to the table syntax in itself, but rather depend on grammars defined by other institutions or services, or on requirements and constraints of semantic nature. In particular, it is necessary to make sure that the mentioned identifiers really exist or can potentially exist, i.e. are (or can be) registered in the service that curates or provides them, and that the bibliographic resource represented in a row of META-CSV contains information that is compatible with OCDM, i.e. falls within the model of data representation adopted by OpenCitations. For example, a CITS-CSV row containing

an identifier for a citing entity that does not seem to exist might be critical or wrong, and would require a double-check by the user; or a META-CSV row corresponding to a bibliographic resource to which a given type has been assigned may have only a certain set of values in the identifier field to be compliant with OCDM.

Starting from the sources defining the two documents' validity and from the considerations above, I have divided the features of the data to be checked by the validator into four categories, which will be at the core of the implementation process and will be treated more detailedly in under the “Validation Process” Section:

1. Format and syntax of the document as prescribed by OpenCitations
2. Externally defined syntax of identifiers
3. Existence of the entities mentioned in the document
4. Semantic conformance with OpenCitations Data Model

Intended outcome

The intended outcome of this work is a software to validate the citation data and bibliographic metadata submitted by users to OpenCitations in the form of tabular documents, in order to be able to ingest this data into CROCI and OpenCitations Meta. The validation process must check that the submitted data conforms to all the applicable rules among the ones included in the four categories mentioned above. The output of the validation process has two main purposes: to be re-used by other software (for tasks like selecting the data to ingest) and to be read and easily understood by users, who should be able to get a deeper understanding of how to produce well-formed and valid documents for submission, get to know exactly which parts of the documents (if any) raise errors or possible ones, and identify the nature of these errors.

The following section describes more exhaustively the methodological approaches taken to implement the software.

Implementation methodology

Input

Both citation and metadata tables have a complex structure. Besides the evident tabular structure, i.e. rows and columns (i.e., fields), there is also a flat structure *inside* the field values (i.e. the cells):

within each field value, there can be one data unit or a collection of multiple data units, divided by a specific separator,³⁴ depending on the table and the field in question. Each of these data units represents the minimal “portion” used by the document to define a specific piece of information, therefore it must first be validated *individually*. Henceforth, these in-field data units will be referred to as “items”. In CITS-CSV, field values that can contain more than one item are the two fields specifying the identifiers of the citing and the cited resource. In META-CSV, the field values that admit multiple items are the ones storing the data of the identifiers, the authors, the venue, the publisher, and the editors. A key factor causing further complexity for the tables’ format is the fact that the cells under certain fields can contain item(s) that are themselves made of smaller components; these components can be of different kinds and each of them must be validated accordingly: in META-CSV, the fields representing entities like responsible agents (author, editor, and publisher) and other bibliographic resources (venue) can contain plain text for the name of the entity as well as, but not necessarily, its identifier or series of identifiers. Items and components of different type, naturally, require different validation rules, meaning that diverse validation rules might be required to validate the whole content of a single field. The following visual aids can be used to understand the abstract representation of the structure of the table.

...	<i>author</i>	<i>pub_date</i>	...
	Peroni, Silvio [orcid:0000.0003-0530-4305 viaf:309649450]; Shotton, David [orcid:0000.0051-5506-523X]	2023-03-13	

Table 1. Two sample table cells of META-CSV, storing the surnames, names and identifiers of the authors of a bibliographic resource and the publication date of the bibliographic resource.

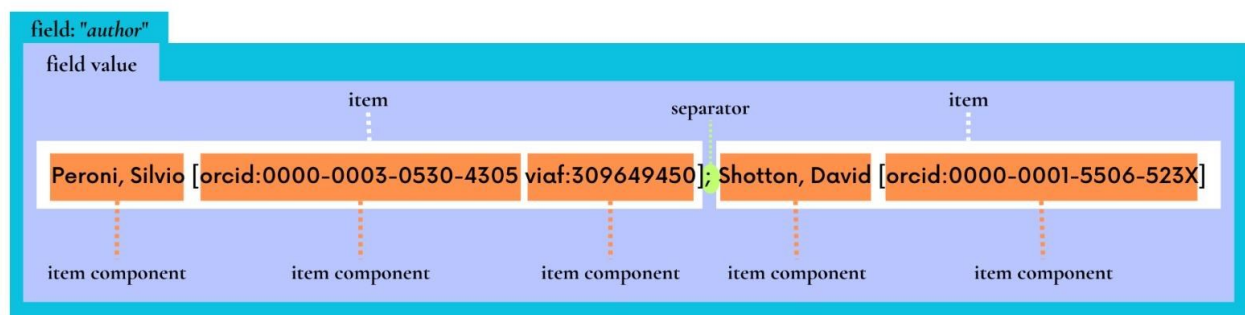


Figure 1. The abstract representation of the internal structure of the table cell containing the data for the author of a bibliography resource (see Table 2). The cell contains two items, each of which corresponds to the entity of an

³⁴ Either a single whitespace character or a semicolon followed by a whitespace character, depending on the field in question.

author; each items has internal components of different kinds (the plain text of the surname and name, and the series of identifiers).

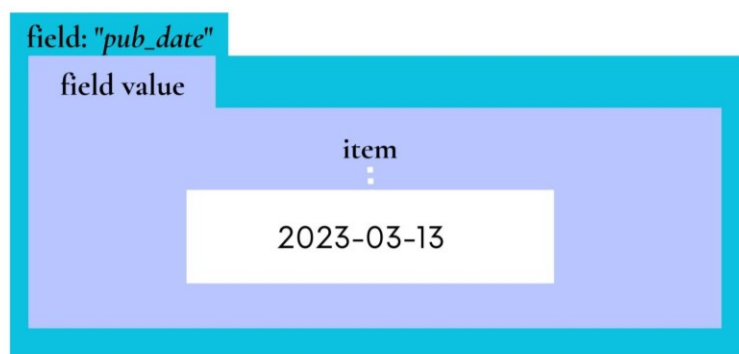


Figure 2. The abstract representation of the internal structure of the table cell containing the data for the publication date of a bibliography resource (see Table 2). The cell contains only one item, which corresponds to the value of the publication date. The publication date field always have a value containing a single item.

In META-CSV, there are three different types of data: plain text (like names, surnames, and titles), date (formatted according to ISO 8601,³⁵ with some possible restrictions), and identifier. In CITS-CSV, there can be only dates and identifiers. Among these datatypes, identifiers are crucial, since they enable to reference and identify real entities from an absolute point of view. In both tables, identifiers follow a precise pattern: they have a prefix specifying the identifier type (e.g. DOI, ISSN, ORCID...), followed by a colon, followed by the identifier's value. For example, a correct way for specifying an identifier would be: “doi:10.1038/502295a”. Considering the importance and the implications they have on the correctness of the data to ingest with respect to OCDM, identifiers require particular attention in the validation process. Moreover, they not only must comply with the “internal” rules defined by OpenCitations in Massari (2022) and Massari and Heibi (2022), regarding the way an identifier should be expressed, but they also need to be expressed correctly according to an “external” syntax, i.e. the one defined by the specific service or agency that provides them, and they must be “real”, existing identifiers. For example, the value of “doi:10.1038/502295a” must comply with the specific identifier scheme of a DOI identifier, and must be registered in the database accessible with the API at “http://doi.org/api/handles”.

Validation Process

The validator must be able to process documents of any size and from any type of user, from the single researcher with little technical experience in data processing, to the large organization capable of processing, even automatically, large amounts of citations and metadata. These features have

³⁵ <https://www.iso.org/iso-8601-date-and-time-format.html>

implications for the design of the implementation process, in particular it poses the following requirements:

1. The software must be able to process potentially very large amounts of data;
2. Errors should be precisely defined and explained, giving their exact location in the document and their identification. This fact is crucial to enable a manual human correction of the errors

To apply the above requirements, the following general principles have been adopted in the software design phase:

1. *Validate the document by applying checks on its smallest parts* (in most cases items, but if applicable, also sub-parts of an item), to ensure the greatest granularity possible and the possibility to precisely identify which parts of the document contain errors or critical characteristics. As will be shown later, however, certain kinds of rules require testing a *relationship* between *multiple* columns (i.e, fields) or rows. To identify with the same precision which pieces of data are critical also in these cases, all the critical sub-parts involved in an error instance will be reported (see section “Output”).
2. *Validate the entire document at each execution of the process*, in order to intercept as many errors as possible to communicate to the user. This means that upon finding an error, the process continues anyway, only stopping when *all* parts of the document have been subjected to the relevant checks; each error is saved and returned in the output along with all other errors found. This process makes it easier for the user to correct errors, since if the process were to stop every time an error is detected, the user would have to intervene by correcting one error at a time, re-submitting the document as many times as many errors were there in the version of the document submitted the first time.
3. *Do not apply multiple checks on an item that has already failed one*, i.e. do not validate a piece of data if it is already known that it will have to be modified by the user. This is to lower the computational cost of the process, ensuring that each time the process is applied to the entire document, checks that would be useless are not executed. This principle, however, only applies in cases where compliance with one rule is a prerequisite for compliance with the other rules for the same item. Otherwise, i.e. if the outcomes of two checks on the same item are mutually independent, both checks are executed straight away (i.e. before the user intervenes with any corrections).

The validation process consists of four steps, in a sequence that mirrors the categorization of the validation rules proposed in section “The data”. Each row, field and item of the document undergoes

validation by being tested on the rules of those categories that are applicable to it. In the case of checks that are applied to *single* items, the first three steps, when relevant,³⁶ are applied to the item sequentially, meaning that if a piece of data fails for a rule at the first step, it will not be checked against the rules that are part of the following ones, since the correctness with respect to one of these levels always depends on the correctness with respect to the previous one.

When it comes to ensuring the correctness of a *relationship* between fields or rows, the corresponding rule is applied regardless of the outcome of the checks on the smaller parts (items) contained in the fields and rows; this is because the applicability of the rule, in this case, might not rely on the validity of the items.³⁷

In summary, the checks regarding the relationships between fields and the relationships between rows are always applied, regardless of the validity of the single items they contain. Conversely, checks on individual items are performed on each item sequentially, so that each check whose outcome depends on the positive outcome of the previous one is executed only if the previous check has been passed.

First level: internal syntax

This category includes the rules that are specified in Massari (2022) and Massari and Heibi (2022): the tests are meant to ensure that the document is well-formed and syntactically valid with respect to the syntax defined in those documents. Therefore, a document for which no error is reported after this step can be processed by the software of OpenCitations, although it is not granted to contain “real” data, i.e. correct and existing identifiers, nor to store information that is compatible with OCDM.

The great majority of the rules in this category apply to individual items. On a CITS-CSV document, for instance, it is verified that the identifier values are preceded by a prefix and that the prefix corresponds to an identifier type managed by OpenCitations software, or that the date is formatted correctly. In META-CSV there are similar rules checking the other fields too. In case of an item of type identifier, failing a rule applied in this step prevents the process to trigger the rules of the following steps on the same item.

³⁶ Of course, some level are only applicable to a subset of the data. More specifically: the first level (compliance with the table syntax) is applied to both tables and to all fields and items; the second level and third level (identifiers syntax and identifiers existence) are applied to both tables but only on the items of identifier type; the fourth level is applied only to META-CSV, since it involves semantic information that is not expressed in CITS-CSV.

³⁷ In case an item involved in the execution of these kinds of “relationship” rule has failed some checks, it is possible, though not certain, that the rule check is not triggered. In such a case, only the error in the item is reported (making the whole row invalid) at the first execution of the validation process, and the relationship rule will be triggered only once the item in question will be corrected by the user, in any following execution.

Other rules are based on the comparison of different fields. In META-CSV, for example, it is necessary to check that the content of a given field is specified, depending on which other fields store content in the same row. This rule defining these relations is included in this category because it is explicitly written in the reference specification for the tables' syntax, and therefore can be intended as a component in the definition of META-CSV's well-formedness. Nonetheless, it also has a significant effect on the *semantic* aspects of the validation, since the absence of required fields in the document would result in ingesting incomplete information.

The first level of the process also includes rules concerning the relation between multiple rows, namely the fact that a bibliographic resource or a citation cannot appear more than once in the document, i.e. no duplicate rows are allowed. Despite not being included in the reference specification, it has been decided to introduce this rule, in order to make sure to ingest only consistent data. Checking the conformance to this rule is based on identifiers: if an ID appears more than once in META-CSV, or a pair of identifiers appear more than once with the same roles of citing and cited entity in CITS-CSV, the bibliographic resource or the citation is duplicated and the corresponding error is reported.

Second level: the external syntax of identifiers

In both tables (CITS-CSV and META-CSV), identifiers play a crucial role in storing information. Therefore, it is crucial to verify their validity, and even more important to justify *why* they are not valid, if it is the case. The second step concerns the validation of the identifier's syntax, therefore it only applies to items that are identifiers; we choose to define two separate steps to check the syntax and the actual existence of an identifier in order to be able to clearly report the reason leading to an item's invalidity, and because it is not always possible to verify whether an identifier exists.

It is worth noticing that the syntax of a given identifier is not defined by OpenCitations, but by the institution or organization that provides it – sometimes, but not always, in an appropriate public documentation. For defining the syntax to which each identifier must conform, it would be best to rely on feasible definitions in official or semi-official documents from the organization curating that identifier; since these documents often could not be found, sometimes other resources have been used, like third-party guidelines and Wikidata.³⁸

Some identifiers were already processable with the existing software in the OpenCitations codebase; specifically, DOI, PMID, ISBN, ISSN and ORCID were already managed, since they are included in the data of the current indexes (COCI, DOCI and POCI). With crowdsourced data, however, other

³⁸ https://www.wikidata.org/wiki/Wikidata:Main_Page

popular and different identifiers might be submitted by users, and therefore must be dealt with, namely PMIDs, Wikidata Q-IDs, Wikipedia page IDs, URLs, ROR IDs, VIAF IDs and Crossref IDs. In order to provide a way to process these new identifiers, and make the software available also for other purposes beyond the validation of crowdsourced data, it has been decided to *extend* the existing software.

The identifiers accepted in OpenCitations crowdsourced tables are 13. Eight of them can be associated with bibliographic resources (therefore can be used in both tables): DOI, ISSN, ISBN, PMID, PMCID, Wikidata Q-ID, Wikipedia page ID, and URL. Responsible agents, i.e. authors, editors, and publishers, can be identified with a ROR ID, a VIAF ID a Crossref ID, an ORCID, or a Crossref ID, which can be all used in META-CSV only (since there are no entities of responsible agents represented in CITS-CSV).

Third level: the existence of identifiers

In addition to being formally correct, in order to fulfill their inherent function identifiers must be externally (and universally) recognized to identify a real-world entity. To verify that a specific identifier “exists”, each identifier in META-CSV and CITS-CSV is queried in the database of the service or organization which provides it. If an ID is accessible from the database of the registration agency providing it, this confirms the fact that the identifier exists. Nonetheless, an identifier might be assigned to a bibliographic resource or responsible agent, without yet being shown as such in the publicly accessible data. For this reason, it has been decided to mark all the cases where an ID does not seem to be registered as “warnings” rather than errors. The difference, treated more in depth in sub-section “Output”, resides essentially on the fact that warnings aim at directing the attention of the user to a specific piece of data, that is likely – but not certain – to contain an error or some kind of inconsistency; errors, on the other hand, prevent the whole row that contains them to be submitted as valid data.

From an implementation perspective, the same approach that has been applied for checking an ID’s syntax has been adopted for this validation step as well: extending the existing software to deal with the new identifiers.

IDs are checked for existence only when they have passed the two preceding validation steps, since it would not be possible to verify the existence of an invalid or “impossible” identifier.

Fourth level: semantics and compliance to OCDM

This step checks whether the data in META-CSV, in addition to being valid and referring to existing entities, is also compliant with OCDM. In fact, this is necessary since the data for a bibliographic resource could pass the previous steps of the validation, therefore be correctly processable by OpenCitations, but still hold inconsistent information from a semantic point of view. For instance, if a specific row indicates in the “type” field that such resource is a *journal article*, but the associated identifier is an ISSN, this semantic relationship is incompatible with the OpenCitations data representation model.

One of the objectives of this work was to attach to the table validation process, whose outcome is primarily aimed at users, a validator that checks data conformity to OCDM using pySHACL. Due to a lack of time, this part has not been accomplished, although it remains a useful integration, as it would cover aspects of the data that are not controlled by the software presented here. In any case, to work around this limitation, it should be sufficient to apply a restriction on the possible identifier-type pairs during the semantic validation process. That is, the definition of an alignment between IDs and types, which describes which IDs can be associated with a bibliographic resource assigned a given type, and vice versa. This strategy, together with what has already been validated in the previous steps, should ensure that semantic information that is inconsistent or incompatible with OCDM is not produced from the data contained in the table.

Output

A significant effort for this work concerns the definition/presentation of the output of the validation process. Indeed, given the purpose of the software, it is paramount to focus on defining what information it should hold and how it should vehiculate it. It is helpful to think about this while taking into consideration two basic principles, namely the fact that the output must be both machine-readable and human-readable. Machine-readability is necessary to ensure that the validated data can be processed by a software, for example for re-elaborating the format of the output, building automatic processes starting from it, re-using the data in a graphical interface, selecting parts of the document that appear to be invalid, etc. Human-readability is fundamental to provide the users with feasible information about what data can be ingested in the indexes, as well as helping them to correct the data they have submitted and understand how to create valid documents for potential future submissions.

The validation output is provided in the form of a report listing all the error instances that can be found by one execution of the process. In order to meet the objectives mentioned above, each of these reported errors stores the following information:

- the position of all the pieces of data that are interested in the error;
- which validation level failed;
- the type of error. i.e. “warning” or “error”, where actual errors make the interested parts invalid, therefore preventing the row containing the error to be ingested, while warnings only signal to the user some potential mistake in order to focus the attention on certain data, but still allows the interested parts to be fully submitted;
- a unique label for the error, which can be used by a machine in order to properly process the output;
- a user message explaining the error and its possible causes in natural language.

Particular attention has been given to finding a feasible way to express the position of the error in the document, so that the data involved in the error could be retrieved and processed for different purposes (e.g. building a graphical interface, excluding invalid rows, etc.) while at the same time grant to the user the possibility to exactly see the single parts of the document involved in the error. The part of the report regarding the error’s position indicates two pieces of information: the exact location of all the single pieces of data involved in that error (relative to the whole document) and whether the error regards a single item, multiple fields, or multiple rows.

Due to the importance of the output, a model of the error report has been defined in JSON Schema, as a means of testing, during the implementation phase, that the output of each check on the data conformed to what was expected.

Implementation: high-level overview

This chapter provides a high-level overview of the implemented software. The first two sections provide an introduction to the OpenCitations Data Model (OCDM) and the `oc_idmanager` Python library: the former is one of the points of reference for defining the validity of the data, and therefore plays a role in the implementation choices that have been taken; the latter is a pre-existing piece of software that has been extended and re-used in the validation software, but is also used in other software within the OpenCitations infrastructure. The third section deals with the structure of the validation software, and describes its classes and the interaction between them, as well as their main functionalities. The last section illustrates how the process is launched by being called in the command line interface.

OpenCitations Data Model

The way in which bibliographic and citation metadata are represented within the OpenCitation datasets is defined in the OpenCitations Data Model (OCDM) (Daquino, Peroni, and Shotton 2020; Daquino et al. 2020). OCDM was created in 2016 to provide a formal description of the data within the OpenCitations Corpus (OCC),³⁹ an open repository of scholarly citation data expressed in RDF. It was later extended and modified in 2019, to cover the need to represent the other datasets developed in the meantime by OpenCitations, as well as the ones emerging from the adoption of OCDM by other organizations. OCDM re-uses a number of terms from other ontologies,⁴⁰ using Semantic Web technology to capture and describe bibliographic metadata: in this way, OpenCitations provides a way to publish this data on the web as Linked Open Data, making it machine-readable and interoperable. As shown in the diagram below (Figure 3), OCDM allows describing a bibliographic resource (class `fabio:Expression`, with subclasses grouping entities according to their type of resource⁴¹) and its embodiment (`fabio:Manifestation`), by capturing metadata such as the publication date (property `prism:publicationDate`) and the title (property `dcterms:title`); it is possible to describe the responsible agents (authors, editors, publisher, contributors...) for that bibliographic resource (class `foaf:Agent`) and associate it to a list of other

³⁹ <https://opencitations.net/corpus>

⁴⁰ These ontologies include the ones in the set of the Semantic Publishing and Referencing Ontologies (SPAR) (Peroni and Shotton 2018b) and well-known ontologies like PROV-O (Sahoo, McGuinness, and Lebo 2013) and the Web Annotation Ontology (Sanderson, Ciccarese, and Van de Sompel 2013). The relevant subsets of the terms used in OCDM are all gathered in the OpenCitations Ontology (OCO): <https://w3id.org/oc/ontology>

⁴¹ Almost all these subclasses (e.g. `fabio:Book`, `fabio:JournalArticle`, `fabio:JournalIssue`, etc.) are taken from the FRBR-aligned Bibliographic Ontology (FaBiO): <https://sparontologies.github.io/fabio/current/fabio.html>.

resources it references (class `biro:BibliographicReference`). In `OpenCitations` citations are represented as first-class entities: a specific citation has its related metadata, like the timespan between the publication date of the citing and cited work, details about the location and textual context in the citing and cited documents, and a unique local identifier associated with the citation itself (Shotton 2018; Heibi, Peroni, and Shotton 2019b).

Besides local identifiers, though, bibliographic resources and responsible agents can be labeled with a public, third-party persistent identifier. Public persistent identifiers link to an entity in the OpenCitations dataset in a way that allows external systems, too, to uniquely identify the entity, granting that the link to this entity is robust, accessible, interoperable, and reusable. In this work, the external identifiers considered are the ones associated with bibliographic resources and responsible agents, i.e. members of the class `datacite:Identifier` in OCDM. To deal with external identifiers, the OpenCitations codebase has a dedicated piece of software, the Python library `oc_idmanager` (described in the next section), a fundamental tool in the development of the validation software.

Figure 3. The Graffoo⁴² diagram of the OCDM, from (Daquino, Peroni, and Shotton 2020).

oc_idmanager library

oc_idmanager is the Python library used by OpenCitations software to perform various operations on external identifiers, processing them in a consistent and uniform way across all the datasets in the infrastructure depending on different data sources

oc_idmanager allows processing an external identifier by instantiating the class that corresponds to it and executing the methods of the class (e.g. a DOI would be processed with the methods in `oc_idmanager.DOIManager`). All ID manager classes, represented in a simplified UML diagram in Figure 4, are instances of the class constructor `IdentifierManager`, from which they inherit the following methods (which share the same general functionalities, but have different implementations, depending on the specific ID):

- `is_valid` assesses the validity⁴³ of the identifier passed as a parameter and optionally, if possible, returns extra information (by setting the `get_extra_info` parameter to `True`) about the entity to which the ID corresponds. This method directly calls the other methods of the class: `syntax_ok` and `normalise` are always called, while the other two are called depending on their applicability (`exist`) or requiredness (`check_digit`).
- `normalise` normalises the ID passed as an argument, either with or without its prefix (i.e. the abbreviation of the identifier type, e.g. “doi:”). It returns the normalised string of the identifier if the process was successful (with or without its prefix, according to the parameter specified in input), `None` otherwise.
- `check_digit` assesses the well-formedness of the input ID by using an ID-specific check digit, if the ID in question has one (it is implemented only for ORCID, ISSN and ISBN). It returns a boolean value.
- `syntax_ok` checks that the identifier is well-formed with respect to the specific syntax of its identifier scheme, verifying that it matches a regular expression pattern. Returns a boolean value.

⁴² <https://essepuntato.it/graffoo/>

⁴³ As identifier is valid if its value conforms to the specific identifier scheme corresponding to the identifier’s prefix (i.e. the syntax of the identifier) and is registered in the corresponding database.

- `exists` checks whether the input identifier is publicly registered in the database of the organization or service by which it is provided, as long as this data is freely accessible. A request is sent to the ID-specific API, and the response is then interpreted to assess whether the identifier actually exists, i.e. is registered. It returns `True` if the ID is registered. An `allow_extra_api` parameter can be optionally specified with a list of additional APIs to retrieve the data from. Since for some IDs it is not possible to freely access the pertinent API, for such IDs (e.g. ISBN) this method is not implemented.
- `extra_info` extracts additional information from the response of the ID-specific API and returns it as a dictionary. Just like `exists`, it is implemented only when an API is freely accessible.

When instantiating an identifier manager (a child class of `IdentifierManager`), it is possible to disable API calls (if they are applicable at all). Moreover, one can pass to the class instance a dictionary storing previously retrieved data, from which to get the outcome of the validation methods (`syntax_ok`, `exists`, and `is_valid`) without the need to process the input ID again.

Originally, `oc_idmanager` included only the identifier managers for the identifier types that were included in the currently implemented indexes (COCI, DOCI, and POCI), i.e. DOI, PMID, ISBN, ISSN, and ORCID. Ingesting crowdsourced data in CROCI and OpenCitations Meta, though, requires a consideration of other additional identifier types, that are likely – due to their widespread use and popularity – to be present in the data submitted by users. For this reason, it has been chosen to extend the `oc_idmanager` library to include new identifier managers for the following types of identifier: PMCID, Wikidata Q-ID, Wikipedia page ID, and URL for bibliographic resources; ROR ID and VIAF ID for responsible agents.

After this extension, it has been possible to re-use `oc_idmanager` in the validation software, with the advantage of re-using a tool that is already known within the OpenCitations infrastructure, tested and efficient. At the same time, the existing tool has been made suitable for managing the new identifiers also within other software of the OpenCitations infrastructure (such as the software for OpenCitations Meta, which will include crowdsourced data). In this way, it will be possible to manage new identifiers exploiting (a subset of⁴⁴) the same functionalities that were already provided for DOI, PMID, ISBN, ISSN, and ORCID, granting consistency and uniformity in dealing with this type of data across the infrastructure and avoiding to recur to new, different solutions that would have to be made compatible with the rest of the software.

⁴⁴ The `extra_info` method has not been implemented, yet, for most of the new identifier managers.

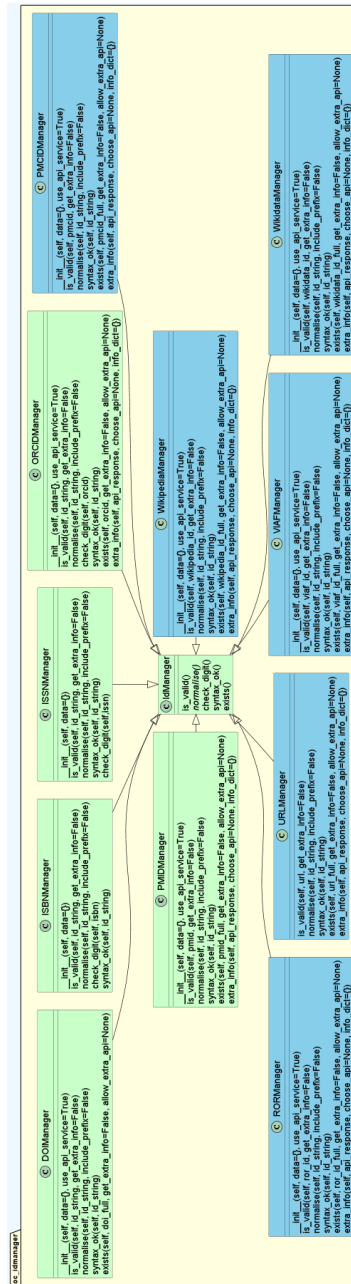


Figure 4. The UML class diagram of `oc-idmanager`. The classes recently implemented to deal with new identifiers are represented by blue boxes. All classes inherit their methods from the base class, `IdManager`, represented by the central box pointed by the arrows.

The validation software: classes and functionalities

This section presents the structure of the Python library developed to validate user data. The next subsections provide a high-level overview of the classes and methods of the library, their relations, and their functionalities.

Validator class

The main class in the software is the `Validator` class. It contains the two methods that are at the core of the validation process (`validate_meta` and `validate_cits`, for the validation of META-CSV and CITS-CSV respectively) determining what validation rules must be checked at any position of the document and stating if, and how, an error report must be created. Upon instantiation, `Validator` is passed two arguments: the path to the input CSV document to validate (`csv_doc`) and the path to the directory where to save the file storing the output of the validation (`output_dir`). The `read_csv` method reads the CSV file as a list of dictionaries, where each dictionary represents a row, and each key-value pair in the dictionary represents a row field and the value it contains; the output list of dictionaries produced by `read_csv` is initialized as an instance attribute (`data`) and reused by the `process_selector` method, which analyses the input data and outputs a string corresponding to either `"meta_csv"` or `"cits_csv"`, indicating which type of table has been passed as input; in case the table is not processable because it is not structured as neither META-CSV nor CITS-CSV, an error is raised. The output of `process_selector` is stored in an instance attribute (`table_to_process`), which is reused inside the `validate` method. `validate` calls one of the two main methods (`validate_meta` and `validate_cits`) according to value of `table_to_process`. The methods described so far grant that the pre-processing of the data, i.e. the set of operations that need to be performed before the phase of the actual validation, takes place automatically just by calling the `validate` method on an instance of `Validator`, without the need to specify other parameters than the input and output paths; this means that the process does not require the user to indicate which table to process, since the table type is automatically detected (as long as the submitted document has valid fields for one of the two types).

The `validate_meta` and `validate_cits` methods work by iterating over the list of dictionaries (each corresponding to a row), and applying checks on the single values of these dictionaries (field values) and the single elements they are composed by (in-field items, such as each of the identifiers in the *id* field in META-CSV, and – when applicable – item components, like each of the IDs associated with a single responsible agent in the *author* field of META-CSV).⁴⁵ For items

⁴⁵ In the present work, “items” are defined as the elements inside a field. Some fields can contain a collection of items, in which case they must be separated by a specific separator. E.g. the *id* field of META-CSV can store multiple identifiers for the same bibliographic resource, and these must be separated by a single whitespace character. Two field values of META-CSV, *author* and *editor*, can store multiple items composed of elements of different types: each item correspond to a responsible agent entity and can contain the plain text of the surname and name of the entity, followed by a list of IDs enclosed in square brackets; items, corresponding to entities, must be separated by a semicolon and a whitespace character, and IDs inside brackets must be separated by a single whitespace character. E.g.: “Berners-Lee, Tim [orcid:0000-0003-1279-3709 viaf:85312226]; Hendler, James A. [viaf:107818627 wikidata:Q6135847]”.

and item components contained in a field value, there is a set of relevant rules that must be checked depending on the type of field: at any given state of the iteration over the field content, specific rules are checked according to the current field. Moreover, there are rules that involve different fields of a single row and rules that involve different rows in the table.

Almost all of these rules are checked by calling validation functions that are methods of other classes, each class corresponding to a validation step: the `Wellformedness` class deals with the first step of the validation, the one concerning the table format and the syntax defined in Massari and Heibi (2022) and Massari (2022); the `IdSyntax` class contains the methods for the second step of the validation, concerning the ID-specific syntax; the `IdExistence` class contains the methods for the third step, checking that an ID is registered; the `Semantics` class checks the compliance to the OCDM.

Whenever a piece of data is found to be invalid, an error or warning is created by calling the `create_error_dict` method from the `Helper` class, and the dictionary returned by this function, representing a single error, is added to the global list of errors to be returned by `validate_meta` or `validate_cits` at the end of the validation process.

`Wellformedness` class

The `Wellformedness` class contains methods for checking the first step of the validation, the one concerning the rules defined in the specification for META-CSV and CITS-CSV tables. At this level, all datatypes (date, plain text, and identifiers) and all table parts (rows, fields and items) of both tables are checked for at least one rule.

The methods included in `Wellformedness` are the following:

- `wellformedness_br_id` checks, using a regular expression, that the format of each item in the *id* field of META-CSV or in the *citing_id* and *cited_id* fields of CITS-CSV is valid, i.e. that a single ID of a bibliographic resource is formed with one of the accepted prefixes for such entity,⁴⁶ followed directly by (any) identifier value. It makes sure that the string of the

There are also two fields, META-CSV's *venue* and *publisher*, that can only have one item, but this can be composed by parts of different types, i.e. plain text for the name of the entity, followed by a list of identifiers enclosed in square brackets and separated by a single whitespace character.

E.g.: "Nature [wikidata:Q180445 issn:0028-0836]"

Other fields – namely *citing_publication_date* and *cited_publication_date* in CITS-CSV, and *title*, *pub_date*, *volume*, *issue*, *page*, and *type* for META-CSV – can contain only one item which does not contain parts of different types (therefore does not require different validation functions for each part).

⁴⁶ The accepted prefixes for bibliography resources in OpenCitations tables are the following: `doi:`, `issn:`, `isbn:`, `wikipedia:`, `wikidata:`, `pmcid:`, `pmid:`, `url:`.

identifier does not contain any illegal characters, such as extra spaces between the prefix and the identifier value, or uppercase letters in the prefix.

- `wellformedness_people_item` checks, by using a regular expression, that the format of a single item in the *author* and *editor* fields in META-CSV is valid.⁴⁷
- `wellformedness_publisher_item` checks, by using a regular expression, that the format of a single item in the *publisher* field of META-CSV is valid. The syntax to check is very similar to the one for “people” entities (*author* and *publisher* fields).
- `orphan_ra_id` is used to apply a further check on the *author*, *publisher*, and *editor* fields in META-CSV, by looking for possible IDs of responsible agents that are not enclosed in brackets, as they should be. If a string that is likely to be an identifier is found outside of the brackets, the method returns False.
- `wellformedness_date` checks, by using a regular expression, the format of the *date* field in META-CSV and of the *citing_publication_date* and *cited_publication_date* in CITS-CSV. The correct format is defined according to ISO 8601.
- `wellformedness_venue` checks that the format of the *venue* field in META-CSV is valid, i.e. it contains plain text, optionally followed by one or more of the accepted identifiers, enclosed in brackets and separated by a whitespace character.
- `orphan_venue_id` does the same as `orphan_ra_id` but with the *venue* field in META-CSV.
- `wellformedness_volume_issue` checks the format of the *volume* and *issue* fields in META-CSV.
- `wellformedness_page` and `check_page_interval` check the value of the *page* field in META-CSV, verifying that it is formatted correctly (`wellformedness_page`) and subsequently trying to make sure that the page interval expressed by the value is reliable

⁴⁷ The format for a single item of the *author* field is described as follows in Massari (2022):

Family Name + “,” + “ ” + *Given Name* + “ ” + “[” + *IDs* + “]”.

The authors’ IDs inside square brackets are indicated using the same structure adopted in the “id” column. e.g. “Peroni, Silvio [orcid:0000-0003-0530-4305]”

If there are no IDs, there will be no square brackets either. The author’s given name is not mandatory.

However, the final comma will be present to indicate the incompleteness of this information.

The accepted prefixes for responsible agents are the following: `orcid:`, `ror:`, `viaf:`, and `wikidata:`.

(it returns False if both start page and end page can be converted into an integer and the end page is lesser than the start page⁴⁸).

- `wellformedness_type` checks, by using a regular expression, that the format of the *type* field in META-CSV is valid, i.e. it matches exactly one of the possible values for that field.
- `get_missing_values` iterates over a row of META-CSV and checks that all the fields that are required to have a value are not empty. Some fields, in fact, are required to have a value depending on a number of conditions, such as the absence of values or the presence of specific values in other given fields. This method returns a dictionary containing the fields that are involved in the error; the dictionary is later re-used inside `validate_meta` to build a dictionary corresponding to the error. Required fields and the conditions upon which they are required are summarised visually in Figure 5.
- `get_duplicate_cits` finds out if a CITS-CSV table contains any duplicate citation and/or any self-citation. It does so by iterating over the table rows and seeing if any bibliographic resource appears both in the `citing_id` and `cited_id` of the same row (self-citation) or if any couple of bibliographic resources appears in more than one row with the same roles of citing and cited entity (duplicate citations). In this process, each bibliographic resource is represented by the set of all the identifiers that co-occur together in the same `citing_id` or `cited_id` field of any row in the table; indeed, in the input arguments of this method, when called inside `validate_cits`, there is the list returned by the `group_ids` method of the `Helper` class (see sub-section `Helper` class). Unlike the methods of this class described so far, `get_duplicate_cits` directly produces a list of error dictionaries, since in order to find duplicates it needs to iterate over the whole table: whenever a duplicate citation or a self-citation is found, an error dictionary is created (using the `create_error_dict` method from `Helper`) and added to the list that is returned as output. This list is then included, within `validate_cits`, in the global list of error reports.
- `get_duplicates_meta` finds duplicate rows in META-CSV, i.e. bibliographic resources that are reported in more than one row. It works basically in the same way as `get_duplicate_cits`, in that it leverages sets of co-occurring IDs to represent bibliographic resources, and it returns a list of dictionaries corresponding to the errors found. The only differences with `get_duplicate_cits` are related to the fact that the latter

⁴⁸ Converting page intervals into pairs of integers is made less trivial by the fact that pages can be expressed with alphanumeric expressions and/or roman numerals.

looks for duplicate *pairs* of bibliographic resources, while `get_duplicates_meta` only searches *single* duplicate bibliographic resources.

id	type	title	author	pub_date	venue	volume	issue	page	publisher	editor
	book	M	O	M						O
	dataset (or data file)	M	O	M						O
	dissertation	M	O	M						O
	edited book	M	O	M						O
	journal article	M	O	M						O
	monograph	M	O	M						O
	other	M	O	M						O
	peer review	M	O	M						O
	posted content (or web content)	M	O	M						O
	proceedings article	M	O	M						O
	report	M	O	M						O
	reference book	M	O	M						O
	book chapter	M			M					
	book part	M			M					
	book section	M			M					
	book track	M			M					
	component	M			M					
	reference entry	M			M					
	book series	M								
	book set	M								
	journal	M								
	proceedings	M								
	proceedings series	M								
	report series	M								
	standard	M								
	standard series	M								
	journal issue	O			M		O			
	journal volume	O			M	O				
1..n	(journal article journal volume journal issue)				M	1				
1..n	(journal article journal volume journal issue)				M		1			

Figure 5. The required field values and the conditions of their requiredness. It illustrates which field values are mandatory if no ID is specified in a specific row, depending on the value of the type field. “M” is an abbreviation for mandatory. Conversely, “O” stands for OR, is always present in pairs, and means that at least one element of the pair is compulsory. The last two lines represent that regardless of whether an ID is specified or not, bibliographic resources of type journal article, journal volume and journal issue must always have values for either volume or issue.

IdSyntax class

The `IdSyntax` class has only one method: `check_id_syntax`. `check_id_syntax` takes as input the string of a single ID (of a bibliographic resource or a responsible agent) and instantiates the class of `oc_idmanager` corresponding to its prefix. From this class instance, the `syntax_ok` method is called and passed the input id. Then the output of `syntax_ok` is returned: True if the identifier conforms to the specific ID scheme, False otherwise.

The `check_id_syntax` method is called in both META-CSV and CITS-CSV for every identifier, be it an item of an *id*, *citing_id*, or *cited_id* field, or included in one of the fields containing mixed-type elements, i.e. the *author*, *editor*, *publisher*, and *venue* fields in META-CSV, which can contain

both plain text and a list of IDs. Nevertheless, this check is executed for an ID (an item or part of an item) only if the previous checks (the first validation level regarding OpenCitations format) have been passed by the ID in question; this grants, for example, that the ID's prefix is present and well-formed. If `check_id_syntax` returns `False`, an error is created and reported in the process of `validate_meta` or `validate_cits`.

IdExistence class

Just like `IdSyntax`, the `IdExistence` class has only one method: `check_id_existence`. It works in the exact same way as `check_id_syntax`, but uses the `oc_idmanager`'s `exists` method instead of `syntax_ok`. This allows verifying whether the input ID is registered in the database of the organization or service providing it, as long as it is freely accessible via a request to its API. If this is not the case, the `exists` method returns `True` by default, therefore no error will be created and reported inside `validate_cits` or `validate_meta`.

Semantics class

The `Semantics` class contains only one method, `check_semantics`, and one instance attribute: the `id_type_dict` dictionary resulting from reading an external JSON file. In this file, shown as a table below (Table 2), each possible type of bibliographic resource, i.e. each of the accepted values in the *type* field of a given row, is associated with a list of possible identifier types that that resource can have in the *id* field. This type-IDs alignment permits to represent semantic relations in the data that are not validated by the other validation steps, covering important features of OCDM (as anticipated in the second chapter (Methodology), in the section “Third level: the existence of identifiers”).

`check_semantics` takes as input a META-CSV row⁴⁹ and `id_type_dict`, and verifies that all the IDs in the *id* field of the row are compatible with the type specified in the *type* field. It is executed (in `validate_meta`) only if the row, the *id* field, and the *type* field have all passed the relevant checks of the first validation step. Nonetheless, it is worth noticing that `check_semantics` is executed regardless of the validity and/or existence of the single ID items, which is assessed later in the process: while we can be sure that all the needed field values are present and well-formed, we cannot be sure about the validity or the existence of the identifiers in the *id* field. The choice to execute `check_semantics` before the 2nd step and 3rd step checks on the single ID items (i.e. without

⁴⁹ Since the semantic relations that need to be validated are expressed only in META-CSV, this check is applied only to the rows of this table, and not to the ones of CITS-CSV.

taking into consideration their outcome) was taken because the correctness of the semantic relation does not depend on the value of the identifier, but only on its type (represented by its prefix). Indeed, as long as only prefixes that are compatible with the type are present in the id field, the semantic relation is deemed valid, and an empty dictionary is returned. Otherwise, `check_semantics` outputs a dictionary specifying which items of the row are involved in the error; this dictionary is then passed to a call of `create_error_dict` inside `validate_meta`.

type	doi	isbn	issn	pmid	pmcid	wikidata	wikipedia	url
book	✓	✓		✓		✓		✓
dataset/data file	✓			✓		✓		✓
disseration	✓	✓		✓	✓	✓		✓
edited book	✓	✓		✓		✓		✓
journal article	✓			✓	✓	✓		✓
monograph	✓	✓		✓		✓		✓
other	✓			✓		✓	✓	✓
peer review	✓			✓	✓	✓		✓
posted content	✓			✓		✓		✓
proceedings article	✓			✓	✓	✓		✓
report	✓	✓		✓	✓	✓		✓
reference book	✓	✓		✓		✓		✓
reference entry	✓			✓		✓	✓	✓

type	doi	isbn	issn	pmid	pmcid	wikidata	wikipedia	url
book chapter	✓			✓		✓		✓
book part	✓			✓		✓		✓
book section	✓			✓		✓		✓
book track	✓			✓		✓		✓
component	✓			✓		✓		✓
book series	✓		✓	✓		✓		✓
book set	✓		✓	✓		✓		✓
journal	✓		✓	✓		✓		✓
proceedings	✓			✓		✓		✓
proceedings series	✓		✓	✓		✓		✓
report series	✓		✓	✓		✓		✓
series	✓		✓	✓		✓		✓
standard	✓	✓		✓		✓		✓
standard series	✓		✓	✓		✓		✓
journal issue	✓			✓		✓		✓
journal volume	✓			✓		✓		✓

Table 2. The type-ID alignment. Each row corresponds to the IDs that can be associated with a bibliographic resource to which a given type has been assigned.

Helper class

The Helper class gathers methods that are used in processes included in the Wellformedness class (`get_duplicates_cits` and `get_duplicates_meta`) and in the Validator class (`validate_meta` and `validate_cits`). These methods are `group_ids`, `create_error_dict`, and `create_validation_summary`.

`group_ids` method

`group_ids` is used to “divide” the bibliographic resources IDs contained in the tables into a list of sets, where each set contains all (and exclusively) the IDs that appear to be associated with the same bibliographic entity, so that no ID can be a member of more than one set. Two IDs are considered to be associated with the same bibliographic entity if they occur together at least once in the same *id* field for META-CSV, and in the same *citing_id* or *cited_id* field for CITS-CSV. This method is used inside `validate_cits` and `validate_meta` to pass its output to the call of `get_duplicates_cits` and `get_duplicates_meta`, respectively.

`create_error_dict` method

The `create_error_dict` method is one of the core elements of this software, since it allows to create the dictionary containing all the information about a single error; because of this, it is called every time an error is found, i.e. whenever a test has failed, from `validate_cits` and `validate_meta` (and also from `get_duplicates_cits` and `get_duplicates_meta`, since they return directly a list of error dictionaries, which within the main validation processes is just included in the global list of errors concerning all the other checks). Every single “error dictionary” created by a call of `create_error_dict` is added to the global list of errors, which is what is returned by `validate_meta` and `validate_cits`. Despite being called so frequently, this method is extremely simple: it just creates a dictionary combining the data passed to it as arguments. Here below is provided a brief explanation of the keys contained in the output of `create_error_dictionary`, i.e. the report for an error (for details about the kinds of errors see also Figure 7):

- `error_type` can take as a value either `error` or `warning`, where an error makes the row that contains it invalid, while a warning is only meant to point out possible critical aspects and locations to the user.
- `validation_level`: indicates which validation step has failed with one of the following values: “`csv_wellformedness`”, “`external_syntax`”, “`existence`” and “`semantics`”.

- `valid` is just an explicit way to clarify whether the error makes the row invalid, without being bound to determine this only on the basis of the value of `error_type`. By default, it is set to `False`, but is always set to `True` when the error type is a “warning”.
- `error_label` stores a unique label identifying the error according to its cause, or rather to what kind of check has been executed; for example, if an error is found checking the format of the *pub_date* field in META-CSV (with the `wellformedness_date` method) the `error_label` key has value “date_format”.
- `message` has as value a string, the human-readable explanation of the error, which is meant to be read and easily understood by users. These user messages are retrieved from a dictionary (`messages`) resulting from reading the “messages.yaml” configuration file, where with each key is associated the text of a specific user message.
- `position` is a dictionary, containing the keys: `located_in` and `table`.
 - `located_in` takes one value among “row”, “field” and “item”, corresponding respectively to an error involving more than one row, an error involving more than one field but only one row, and an error involving any number of items but only one field and only one row.
 - `table` is probably the most crucial component of the dictionary, since it permits, in a machine-readable way, to locate in the validated document which values are involved⁵⁰ in the represented error. The value of the `table` key is a dictionary representing the document as a tree structure: the root of the tree is the dictionary itself and its children are the indexes of the rows that are involved in the error; each row branch has as many children as the fields that are involved in the error, and each of these is labeled with the name of the field. Each field node’s value is either a list of integers or `None`: if it is a list of integers, each integer represents the position of an involved item inside the field; if it is `None`, it means that an item should be present, but is missing instead.

In reason of its importance, it is worth dedicating particular attention to the `table` dictionary nested inside the output of `create_error_dict`. In the `table` dictionary, the position of the invalid content inside the cell is always represented as an integer inside a list, which is the data structure used

⁵⁰ Rows, fields and items are involved in an error either because they are invalid by themselves or because they concur to cause an error in reason of their combination with one another. Whether multiple fields or multiple rows are involved is specified by the value of `located_in`.

to represent a “container” of the elements inside the cell itself. Indeed, this way of representing the path to the location of invalid content in the document permits to maintain and reuse the same structure for different cases. By this logic, field values that contain only one item (*citing_publication_date* and *cited_publication_date* in CITS-CSV; *title*, *pub_date*, *venue*, *volume*, *issue*, *page*, *type*, and *publisher* in META-CSV) will always be represented as a list containing either the integer 0 (representing the first position in the cell, which in this case is the only possible position, since there are no other items) or `None` (in case the cell does not have any content). On the other hand, field values that can contain multiple items divided by a specific separator (*citing_id* and *cited_id* in CITS-CSV; *id*, *author*, and *editor* in META-CSV), will be represented as lists containing all the integers equivalent to the positions of the items in the cell that are involved in the error, or `None` if the cell is empty. The variety of all the possible configurations for the `table` dictionary can be illustrated by means of the following examples of error reports, built assuming that they are part of the output of the process validating META-CSV.

Example 1: Error in the format of the *pub_date* field

```
{
  "validation_level": 'csv_wellformedness',
  "error_type": 'error',
  "message": "The pub_date field is not formatted correctly.
Valid formats are YYYY-MM-DD, YYYY-MM, or YYYY",
  "error_label": 'date_format',
  "located_in": 'item',
  "table": {
    0: {
      "pub_date": [0]
    }
  }
}
```

The dictionary in Example 1 represents an error raised in the first validation level, concerning the syntax of META-CSV (`"validation_level": 'csv_wellformedness'`): in the first row (`"table": {0: {...}}`, where the key in the dictionary is 0 since the position of a row in the table is represented by the index of the row dictionary in the list) the value of the *date* field, which can contain only one item (`"pub_date": [0]`), is not represented correctly. All the data involved in this kind of error is contained in the same field, therefore the error is at the “item” level: `"located_in": 'item'`.

Example 2: Error in the syntax of an ID in the *id* field

```
{
  "validation_level": 'external_syntax',
  "error_type": 'error',
  "message": "The ID does not conform to its identifier scheme",
```

```

    "error_label": 'br_id_syntax',
    "located_in": 'item',
    "table": {
        3: {
            "id": [2]
        }
    }
}

```

The error regards the syntax of an ID: "validation_level": 'external_syntax'. The table part involved in the error is the third item (i.e. the element with index 2 inside the list representing the field value) of the *id* field of the fourth row ("table": {3:{“id”:[2]}}).

Example 3: Duplicate ID inside the *id* field

```

{
    "validation_level": 'cav_wellformedness',
    "error_type": 'error',
    "message": "The same ID is specified twice in the 'id' field",
    "error_label": 'duplicate_id',
    "located_in": 'item',
    "table": {
        2: {
            "id": [0, 3]
        }
    }
}

```

The error consists of the same ID being present more than once inside the same table cell. The items involved are the first and fourth items in the value of the *id* field of the third row:

```
"table":{2:{“id”:[0,3]}}.
```

This tree structure for representing the path of invalid data in the document is suitable also for errors that involve more than one field and/or more than one row. The following examples show how error instances that are caused by pieces of data in different fields of the same row (Example 4) and in different rows (Example 5) would be represented in the error dictionary.

Example 4: Missing required value for *title* of a resource of type “book”.

```

{
    "validation_level": 'csv_wellformedness',
    "error_type": 'error',
    "message": "A 'book' resource requires to specify 'title',
'pub_date', 'author' and 'editor' if no value for 'id' is
specified.",
    "error_label": 'required_fields',
    "located_in": 'field',
    "table": {
        2: {
            "id": [None],
            "type": [0],

```

```

        "title": [None]
    }
}

```

Since the value of the *id* field of the third row is not specified (`"id": [None]`), according to the value specified in the *type* field (`"type": [0]`), namely “book”, the *title* field cannot be empty (`"title": [None]`). Since the error involves items contained in different fields of the same row, the value of `"located_in"` is `"field"`.

Example 5: Duplicate rows (IDs of the same bibliographic entity appear in more than one row).

```

{
  "validation_level": 'csv_wellformedness',
  "error_type": 'error',
  "message": "The same bibliographic entity is represented in
more than one row.",
  "error_label": 'duplicate_br',
  "located_in": 'row',
  "table": {
    2: {"id": [0,1]}
    8: {"id": [0,1,2]}
  }
}

```

The items in the *id* field of the third row and the ninth row of the table are mapped to the same bibliographic entity, which is therefore represented in two different rows. Since the error involves multiple rows the value of `"located_in"` is `"row"`.

`create_validation_summary` method

The `create_validation_summary` method permits to create a human-readable “summary” of the information contained in the list of dictionaries that represent the errors. The resulting validation report is written in natural language and gives an account of:

- the number of occurrences of each type of error (error/warning);
- how many errors were found for a specific rule (based on the `error_label` keys of the dictionaries);
- an explanation of each kind of error that has been found, related to the rule for which the document has failed the test;
- all the locations of each error, grouped by the rule that has been failed (based on `error_label`).

Rather than a summary, the output of this method is meant to be a way to present the validation outcome in a user-friendly way, by structuring it in a fashion that is easier to read and understand for humans. `create_validation_dict` is called within `validate_meta` and `validate_cits`, and its output is saved in an external file in the same directory as the JSON file storing the machine-readable output (i.e. the output of `validate_meta` and `validate_cits`).

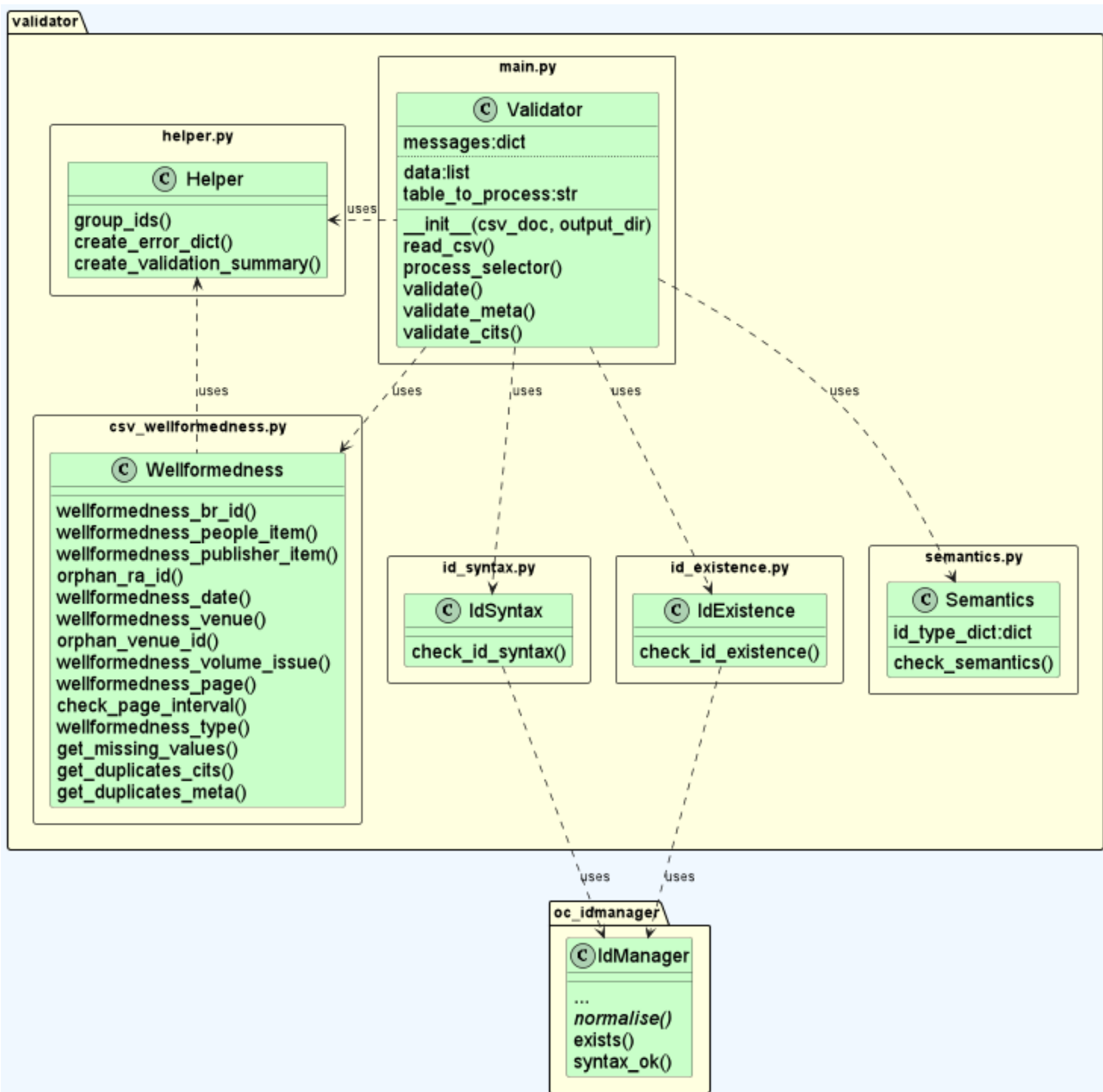


Figure 6. A simplified UML class diagram of the validation software. The dotted arrows represent the fact that the class from which the arrow starts uses methods of the class to which the arrow is pointing. As can be since the structure is flat (i.e. there are no child classes, excepts for the ones in oc-idmanager). Moreover, the arrows pointing to the IdManager class are represented in this way only for graphical reasons: they should actually point to each of the child classes of IdManager, represented in figure 4.

Code launch

The validation process can be launched from the command-line interface (CLI) with a command of the following form, from the parent directory of the `validator` package:

```
python -m validator/main.py -i <path to input csv file> -o <output directory>
```

The following script, in `validator.main` module, permits to call the execution of the software from the CLI.

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--input', dest='input_csv',
required=True, help='The path to the CSV document to validate.', type=str)
    parser.add_argument('-o', '--output', dest='output_dir',
required=True, help='The path to the directory where to store the output
files.', type=str)
    args = parser.parse_args()
    v = Validator(args.input_csv, args.output_dir)
    v.validate()
```

Script 1. Launching the software from the CLI.

This script allows the execution of the validation process from the CLI by using the `argparse` library to parse the command-line arguments passed to the script. The `-i` or `--input` argument is used to specify the path to the CSV document to be validated, and the `-o` or `--output` argument is used to specify the path to the directory where the output files will be stored. The script creates an object of the `Validator` class, passing the values of the `input_csv` and `output_dir` arguments as parameters, and then calls the `validate` method on the `Validator` object to perform the validation process. The `Validator` class, in fact, requires two mandatory arguments upon instantiation: the path to the input CSV file to validate, and the path to the output directory.⁵¹ Once the input document (an instance of either one of the CSV document types) is processed, the output of the validation is stored in two files in the specified output directory: a JSON file containing a list of objects, each of which corresponds to the information about an error, expressed in a machine-readable (and, to an extent, human-readable) way; and a .txt file describing in plain English the outcome of the validation, containing the information listed in section `create_validation_summary` method.

⁵¹ If the output directory does not exist in the system yet, it is automatically created within both `validate_cits` and `validate_meta`.

error_label	Description	validation_level	located_in	field	Document
duplicate_id	An ID is repeated more than once inside a single cell	csv_wellformedness	item	citing_id, cited_id or id	both
self-citation	The same bibliographic entity is found in both citing_id and cited_id of the same row	csv_wellformedness	field	citing_id and cited_id	cits-csv
duplicate_citation	The same citation (a bibliographic entity citing another bibliographic entity) is found in more than one row	csv_wellformedness	row	citing_id, cited_id or id	cits-csv
extra_space	Inside the id field, spaces are found at the beginning of the string or at the end, or extra spaces are found between two IDs	csv_wellformedness	item	citing_id, cited_id or id	both
br_id_format	The single ID (item) is not well-formed according to OC syntax (i.e.: in lowercase, presenting the accepted prefix followed by colon and other non-space characters)	csv_wellformedness	item	citing_id, cited_id or id	both
required_value_cits	The value of a mandatory field (either citing_id or cited_id) is an empty string	csv_wellformedness	field	citing_id or cited_id	cits-csv
date_format	The date is badly formatted (i.e.: It does not match the regex defining any correct possibility)	csv_wellformedness	item	citing_publication_date, cited_publication_date or pub_date	both
uppercase_title	The characters of the title string are all in uppercase	csv_wellformedness	item	title	meta-csv
people_item_format	Any format error in 'author' or 'editor' field	csv_wellformedness	item	author or editor	meta-csv
publisher_format	Any format error in 'publisher' field	csv_wellformedness	item	publisher	meta-csv
orphan_ra_id	A string likely to be a responsible agent ID or a sequence of such is found NOT enclosed in brackets	csv_wellformedness	item	author, editor or field	meta-csv
duplicate_br	The same bibliographic entity is represented in more than one row	csv_wellformedness	row	id	meta-csv
venue_format	Any format error in 'venue' field	csv_wellformedness	item	venue	meta-csv
orphan_venue_id	A string likely to be an ID for a venue or a sequence of such is found NOT enclosed in brackets	csv_wellformedness	item	venue	meta-csv
volume_issue_format	Any format error in 'volume' or 'issue' field	csv_wellformedness	item	volume and 'issue'	meta-csv
page_format	Any format error in 'page' field	csv_wellformedness	item	page	meta-csv
page_interval	The interval in 'page' field is invalid or it is impossible to convert it into integer numbers	csv_wellformedness	item	page	meta-csv
type_format	Any format error in 'type' field	csv_wellformedness	item	type	meta-csv
required_fields	One or more required values for certain fields are not specified in the row, OR the specified values in 'type' are incorrect (only for rows with 'volume' or 'issue' specified)	csv_wellformedness	field	(all conditionally involved fields)	meta-csv
br_id_syntax	Error in the externally-defined syntax of the identifier for a bibliographic resource	external_syntax	item	id, venue	both
br_id_existence	the ID of the bibliographic resource is not registered in the service accessed via API	existence	item	id, venue	both
ra_id_syntax	Error in the externally-defined syntax of the identifier for a responsible agent	external_syntax	item	author, editor, publisher	meta
ra_id_existence	The ID of the responsible agent is not registered in the service accessed via API	existence	item	author, editor, publisher	meta
row_semantics	the ID of the bibliographic resource is not compatible with the value of the 'type' field according to OCDM	semantics	field	id, type	meta

Figure 7. An abridged version of the taxonomy of errors. The errors are represented by the corresponding “error_label”, accompanied by a description of what causes the error (only for explanation purposes), the corresponding validation level, the value of “located_in” and the fields that might be involved in the error (and consequently be reported in the table dictionary nested in the output dictionary), and the documents that might be concerned by the error.

Implementation: technical details

This chapter presents the validation process in detail, providing a technical, low-level explanation of the various components of the software and analyzing the code of some of its parts. The first section provides an overview of the general features of the software and its imports, the second presents details on the methods of the `oc_idmanager` library that are used in the validation software. Then, the validation process is explained: first, the pre-process phase, then the process for validating META-CSV tables is illustrated, including an explanation of the process for creating the errors and of the processes executed for each validation level. Finally, the last section briefly presents the process for validating CITS-CSV.

General features

The validation software is implemented in Python 3.9. The table below (Table 3) illustrates the imported external libraries and built-in modules, the general context of their use, and their usage/role within the software. The table does not include the libraries that are used in `oc_idmanager`,⁵² which itself requires the installation and import of a set of external libraries.

Library/module	Elements imported	Context	Use of the imported element
<code>argparse-1.4.0</code>	<code>ArgumentParser</code>	<code>main.Validator</code>	Used to enable the call from the CLI, by passing the arguments to an instance of the <code>ArgumentParser</code> class.
<code>csv</code>	<code>DictReader</code>	<code>main.Validator</code>	Reads the CSV input file into a list of dictionaries.
<code>json</code>	<code>load</code>	<code>main.Validator</code>	Loads the JSON file storing the type-ID alignment (“ <code>id_type_alignment.json</code> ”) into a dictionary, to pass it to a call of <code>check_semantics</code> method from <code>Semantics</code> class.
	<code>dump</code>	<code>main.Validator</code>	Writes the output of the validation process into a

⁵² https://github.com/opencitations/identifier_manager

Library/module	Elements imported	Context	Use of the imported element
			JSON file.
oc_idmanager	doi, isbn, issn, orcid, pmcid, pmid, ror, url, viaf, wikidata, wikipedia	id_existence.IdExistence	A subclass of IdManager is instantiated (e.g. DOIManager) to pass the string of the ID to syntax_ok or exists methods.
		id_syntax.IdSyntax	
os	makedirs	main.Validator	Used to create the directory where to store the output files in case the filepath does not point to a real directory.
os.path	exists	main.Validator	Used to check if an output directory already exists.
	join		Used to join the path of the output directory specified as argument of the class instance and the name of the output files.
pyyaml-6.0	full_load	main.Validator	Used to read as a dictionary the YAML configuration file storing the user messages
re	finditer	main.Validator	Used to find all the elements of identifier type inside a single (mixed-type) item of the <i>author</i> and <i>editor</i> fields in META-CSV.
	match	csv_wellformedness.Wellformedness	Used to validate a string of any item or field against a regex pattern
	search	csv_wellformedness.Wellformedness	Used to see if there is any substring in an <i>author/editor</i> field that is likely to be an ID, but is not represented as such
	sub	csv_wellformedness.Wellformedness	Used in combination with finditer find all the elements of identifier type inside a single (mixed-type) item of the <i>author</i> and <i>editor</i> fields in META-CSV

Library/module	Elements imported	Context	Use of the imported element
roman 3.3	fromRoman InvalidRomanNumeralError	csv_wellformedness.Wellformedness	fromRoman is used to convert numbers expressed as Roman numerals into integers, so to compare start and end page of META-CSV's <i>page</i> field. In the same context InvalidRomanNumeralError is imported to be caught and managed.

Table 3. The imports and the dependencies of the validator. Built-in modules are formatted in italics.

Identifier Managers

This section explains the low-level features of the `oc_idmanager` Python library, which is used to process public identifiers across the OpenCitations infrastructure and has been expanded and integrated into the validation software to perform the operations for validating the syntax of IDs and verify their existence. Besides data validation, this library also provides functionalities for information retrieval. The `extra_info` method could be defined for any identifier, to obtain and store metadata about the entity it represents. However, since the purpose of the software described in the present work is to validate data, in extending `oc_idmanager` with new classes to deal with the identifiers that were not managed yet, I focussed on implementing, for each class, the methods aimed at data validation, leaving the implementation of the `extra_info` method for the newly added identifiers among the possible future developments for this software.

The procedure described here concerns the use of `oc_idmanager` for the validation of IDs, with regard to their compliance with the specific identifier scheme (i.e. the ID's syntax) and to their being publicly registered as such in the database of the organization providing them (i.e. the ID's existence). In order to explain this validation procedure, the example of one of the classes of `oc_idmanager` is considered and illustrated, namely the `VIAFManager` class, in the `oc_idmanager.viaf` module. The `VIAFManager` class is a child class of `IdManager` class, like all the other ID managers. It has two parameters: `data`, initialized as an empty dictionary, and `use_api_service`, defaulting to `True`. `data` can be set to a dictionary storing data about the validity of identifiers, where keys are identifiers and values are dictionaries including the `valid` key, which itself stores `True` if the ID is valid, and `False` if it is not. The possibility to retrieve data from such a dictionary enables, for instance, to avoid validating an identifier that has already been validated. The `use_api_service`, if set to `False`, prevents the `exists` method to make a request to the VIAF API, being particularly useful

when this is not needed or suitable. Both `data` and `use_api_service` are stored as instance attributes: `_data` and `_use_api_service` respectively. Moreover, the constructor initializes the `_p` instance variable to the string with which a VIAF identifier must be prefixed in the OpenCitations infrastructure (i.e. its prefix, “viaf:”) and the `_api` variable to the string of the URL of the API for accessing the database of the Virtual International Authority File (VIAF).⁵³

```
def __init__(self, data={}, use_api_service=True):
    """VIAF manager constructor."""
    super(VIAFManager, self).__init__()
    self._api = "http://www.viaf.org/viaf/"
    self._use_api_service = use_api_service
    self._p = "viaf:"
    self._data = data
```

Script 2. The VIAFManager class initializer

`VIAFManager` has five methods: `is_valid`, `normalise`, `syntax_ok`, `exists`, and `extra_info`.

In case the expected outcome of the validation process is merely to know whether an ID is valid or not (i.e. if there is no need of knowing, in case the ID is not valid, what caused it to be invalid), then it is possible to use directly the `is_valid` method, since it calls the other two validation methods (`syntax_ok` and `exists`) and returns `True` if both these methods return `True`, and `False` otherwise. In this way, it returns the output of the *global* validation process: in case of invalid data, it is not possible to tell at which step of this process the input ID does not validate. In order to understand the reasons of the invalidity of a datum, it is necessary to clearly distinguish which step, i.e. which validation methods, caused the datum to fail validation. For this reason, as anticipated in the two previous chapters, in the validation software presented here the `syntax_ok` method and the `exists` method are called directly, with two separate (and sequential) calls.

`is_valid` takes as input the ID string and the `get_extra_info` parameter, defaulting to `False`. First, the ID string is normalised (using the `normalise` method), and if the resulting string does not pass the syntax check, `is_valid` directly returns `False`. Otherwise, it checks whether the ID is already stored in `_data`. If it is not there, it proceeds to check the value of the `get_extra_info` parameter: if it is `True`, then the method returns the output of `exists` and a dictionary containing additional information; if it is set to `False`, as by default, it returns only the output of `exists`; in both cases, before returning the output, the key in `_data` corresponding to the identifier is updated with

⁵³ <https://viaf.org/>

the values of the output itself: a boolean value and the extra information dictionary if `get_extra_info` is set to `True`, or only the boolean value if it is set to `False`. In case the identifier is already present in `_data`, `is_valid` directly returns the values cached there: either only the boolean or the boolean and the dictionary with extra information, according to the value set to the `get_extra_info` parameter.

```
def is_valid(self, viaf_id, get_extra_info=False):
    viaf_id = self.normalise(viaf_id, include_prefix=True)

    if viaf_id is None or not self.syntax_ok(viaf_id):
        return False
    else:
        if viaf_id not in self._data or self._data[viaf_id] is
None:
            if get_extra_info:
                info = self.exists(viaf_id,
get_extra_info=True)
                self._data[viaf_id] = info[1]
                return (info[0] and self.syntax_ok(viaf_id)),
info[1]
            self._data[viaf_id] = dict()
            self._data[viaf_id]["valid"] = True if
(self.exists(viaf_id) and self.syntax_ok(viaf_id)) else False
            return self._data[viaf_id].get("valid")
        if get_extra_info:
            return self._data[viaf_id].get("valid"),
self._data[viaf_id]
        return self._data[viaf_id].get("valid")
```

Script 3. The `is_valid` method of `VIAFManager`.

The `normalise` method takes as input the string of the VIAF ID and a parameter (`include_prefix`) specifying whether the string includes the prefix or not, which defaults to `False`. It then tries to normalise the string by removing any character that is not a digit (since a VIAF ID is composed of digits only). It returns either the normalised string of the ID (with or without the prefix, according to the value of `include_prefix`) if the normalisation is successful, or `None` if any error was raised in the processing.

The `syntax_ok` method takes as input the string of the VIAF ID. First, it checks if the prefix is included, and in case it is not, this is added at the beginning of the string. Then, the method checks if the input string matches a regular expression pattern, representing the valid syntax of a VIAF

identifier. If it matches, it returns True; otherwise, it returns False. This method is called by the `check_id_syntax` in the `IdSyntax` class of the validation software. Obviously, the regex pattern depends on the specific identifier scheme. For example, the VIAF identifier consists of a positive number of a maximum of 22 digits, which is represented with the following regular expression: `^viaf:[1-9]\d{1,21}$`. Regular expressions are employed in the `syntax_ok` methods of all the `IdManager` classes. Moreover, they cover a crucial role in many of the methods in the `validator.Wellformedness` class, as they are widely used to check the format of items and fields at the first step of the validation process.

```
def syntax_ok(self, id_string):
    if not id_string.startswith("viaf:"):
        id_string = self._p + id_string
    return True if match(r"^viaf:[1-9]\d{1,21}$", id_string)
else False
```

Script 4. The `syntax_ok` method of `VIAFManager`.

The `exists` method is used to check if a given VIAF identifier exists in the VIAF database. It takes as input the string of the VIAF ID (either with or without the prefix) and a boolean (`get_extra_info`), defaulting to False and indicating whether it should return – together with the boolean value for the existence of the ID – a dictionary with additional information as well. The method first checks whether the `use_api_service` flag (i.e. the instance attribute) is set to True. If it is, a GET request is made to the VIAF API to retrieve information about the input VIAF identifier. Before the request `exists` calls `normalise` on the ID string (with `include_prefix` set to the default value of False, therefore removing the prefix if it is included in the string), in order to format it so that it can be used in the API request. If the API request returns a status code of 200 (i.e. if the request is successful), the method parses the JSON response to see if it contains the given identifier. If the identifier is present, the method returns True, otherwise it returns False. If the API request returns a status code between 400 and 500 (client error), the method returns False, indicating that the given identifier is invalid. If the API request times out or encounters a connection error, the method will wait for 5 seconds and try again, up to three times. If all three attempts fail, the method will return False. If the `use_api_service` flag is set to False, `exists` does not make a request to the API, and directly returns: True, if `get_extra_info` is set to False; or True and the dictionary to add to `_data` (`{"valid": True}`), if `get_extra_info` is set to True. For `VIAFManager`, in fact, the `extra_info` method to retrieve extra information from the API has not been implemented. Therefore, when the `get_extra_info` flag is set to True, the returned dictionary contains one key-value pair indicating only that the ID is valid, while for classes that already have an `extra_info` method this dictionary is the actual result of `extra_info`, and should therefore contain more information.

```
def exists(self, viaf_id_full, get_extra_info=False, allow_extra_api=None):
    valid_bool = True
    if self._use_api_service:
```

```

        viaf_id = self.normalise(viaf_id_full)
        if viaf_id is not None:
            tentative = 3
            while tentative:
                tentative -= 1
                try:
                    r = get(self._api + quote(viaf_id) + '/viaf.json',
headers=self._headers, timeout=30)
                    if r.status_code == 200:
                        r.encoding = "utf-8"
                        json_res = loads(r.text)
                        if get_extra_info:
                            extra_info_result = {}
                            try:
                                result = True if json_res['viafID'] ==
str(viaf_id) else False

                                extra_info_result["valid"] = result
                                return result, extra_info_result
                            except KeyError:
                                extra_info_result["valid"] = False
                                return False, extra_info_result
                        try:
                            return True if json_res['viafID'] == str(viaf_id)
else False

                            except KeyError:
                                return False
                        elif 400 <= r.status_code < 500:
                            if get_extra_info:
                                return False, {"valid": False}
                            return False
                    except ReadTimeout:
                        # Do nothing, just try again
                        pass
                    except ConnectionError:
                        # Sleep 5 seconds, then try again
                        sleep(5)
                valid_bool = False
            else:
                if get_extra_info:
                    return False, {"valid": False}
                return False

        if get_extra_info:
            return valid_bool, {"valid": valid_bool}
        return valid_bool

```

Script 5. The exists method of VIAFManager.

The validation software

This section describes the validation software in detail, explaining the general workflow: the data flow, the validation process (including the integration with oc_idmanager), and the production of the output.

All the files dealing with the processing of the input document are inside a package named `validator`, which itself contains five modules (`main`, `csv_wellformedness`, `id_syntax`, `id_existence`, and `semantics`) each of which contains one only class: `main` contains the `Validator` class, containing the methods to start the process and using the validation methods of the other classes; the rest of the modules contain, respectively, the classes `Wellformedness`, `IdSyntax`, `IdExistence` and `Semantics`. The `validator` package also includes two configuration files: the YAML file “`messages`”, used for retrieving the user messages to include in the validation report, and the JSON file “`id_type_alignment`”, used to look up which identifier types can be legally associated to a given type of bibliographic resource (see section “`Semantics`”).

Pre-process operations

This section describes the operations performed on the data before the steps of the process performing actual validation; these operations include making the data in the CSV document accessible and processable in Python, selecting which process should be executed for the submitted data, and finally calling the execution of that process. All these pre-process operations, like the processes themselves, are dealt with by methods of the `Validator` class.

```
def __init__(self, csv_doc:str, output_dir:str):
    self.data = self.read_csv(csv_doc)
    self.table_to_process = self.process_selector(self.data)
    self.helper = Helper()
    self.wellformed = Wellformedness()
    self.syntax = IdSyntax()
    self.existence = IdExistence()
    self.semantics = Semantics()
    self.messages = full_load(open('validator/messages.yaml',
'r', encoding='utf-8'))
    self.id_type_dict = load(
        open('validator/id_type_alignment.json',
'r',
encoding='utf-8')) # for ID-type alignment (semantics)
    self.output_dir = output_dir
    if not exists(self.output_dir):
        makedirs(self.output_dir)
```

Script 6. The initializer of the class `Validator`.

`Validator` is initialized with two required parameters, both of type string: `csv_doc`, i.e. the path to the input CSV file to validate, and `output_dir`, i.e. the path to the directory where to put the files where the validation output will be stored. Upon instantiation, the `output_dir` argument is directly assigned to an instance variable of the same name, while the `csv_doc` argument is passed to the

`read_csv` method. This method accesses the document via the input file path, opens it in reading mode (using UTF-8 encoding), and passes a copy of it to the `DictReader` constructor; `DictReader` returns an iterable of dictionaries, where each dictionary corresponds to a table row, keys correspond to column names and values correspond to field values. `DictReader` is used with its default settings: the comma character (,) is interpreted as a separator between fields, and dictionary keys are automatically inferred from the first row of the CSV file, which is therefore interpreted as the table header. The `DictReader` iterator is type-casted into a list (of dictionaries) and assigned to the `data` instance variable.

```
def read_csv(self, csv_doc):
    with open(csv_doc, 'r', encoding='utf-8') as f:
        data_dict = list(DictReader(f))
    return data_dict
```

Script 7. The `read_csv` method of `Validator`.

`data` is passed as input to the `process_selector` method, whose output string is then assigned to the `table_to_process` instance variable. `process_selector` iterates over all the dictionaries in `data` and checks that all the keys of each row are exactly compatible with the column names of either META-CSV or CITS-CSV: the string of each dictionary key must match exactly the corresponding column name in one of the two table types; the view object⁵⁴ corresponding to the dictionary keys is type-casted into a list, so that it is possible to compare both the sequence of column names (which is also represented as a list) and the sequence of dictionary keys as ordered data structures. If `data` is recognized to be compatible with the expected input format of one of the two processes, `process_selector` returns a string (either `'meta_csv'` or `'cits_csv'`) indicating which table has been submitted, so that it will be possible to decide automatically which (table-dependent) validation process must be executed. If any incompatibility is found, (e.g. if a dictionary key is in the wrong letter case or is missing entirely for any of the rows), `process_selector` will return `None`.

```
def process_selector(self, data: list):
    process_type = None
    try:
        if all(list(row.keys()) == ["id", "title", "author",
                                   "pub_date", "venue", "volume", "issue", "page", "type",
```

⁵⁴ Python 3 documentation states: “The objects returned by `dict.keys()`, `dict.values()` and `dict.items()` are view objects. They provide a dynamic view on the dictionary’s entries, which means that when the dictionary changes, the view reflects these changes. Dictionary views can be iterated over to yield their respective data, and support membership tests.” <https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects>

```

                                "publisher", "editor"] for
row in data):
    process_type = 'meta_csv'
    return process_type
    elif      all(list(row.keys()))      ==      ['citing_id',
'citing_publication_date', 'cited_id', 'cited_publication_date']
        for row in
            data):
                process_type = 'cits_csv'
                return process_type
    else:
        return process_type
except KeyError:
    print('The submitted table cannot be read as neither
META-CSV nor CITS-CSV')
    return process_type

```

Script 8. The process_selector method of Validator.

The output of process_selector, i.e. "meta_csv" or "cits_csv" or None, is assigned to the table_to_process instance variable.

All the steps illustrated so far take place at the initialization phase, as soon as an object of the class Validator is instantiated. To start the validation process, the validate method of a Validator instance must be called. validate takes as input the variable table_to_process, and according to its value calls the execution of either validate_cits (if the value of table_to_process is "cits_csv") or validate_meta (if the value of table to process is "meta_csv") and then returns its output; if table_to_process is None, it returns a message explaining that the input cannot be processed. The two fundamental methods that perform the actual validation, validate_cits and validate_meta are explained in the following sections.

Validating META-CSV

This section illustrates the process of validating the data read from a META-CSV document as a list of dictionaries. At the core of this process, there is the Validator.validate_meta method, which uses all the validation functions imported from the other classes of the software and executes them on the elements of data they are applicable to, creating an error dictionary if the validation fails.

To check each single structural component of the table (row, field and item) for the validation rules that apply to it, `validate_meta` processes the document by iterating over the data with nested for-loops. It always iterates the dictionaries in `data` (rows), traversing the table vertically, and the key-value pairs in the dictionaries (fields), traversing each row horizontally. Depending on the key of the key-value pair, i.e. depending on which field the current loop variable equals to, it can create a list of substrings of the value (items) from splitting the value of the key-value pair by a specific separator, traversing horizontally a single table “cell”. Finally, depending again on the key of the current key-value pair, it can create a list of substrings from splitting the current item and iterate on it, traversing horizontally a specific “section of a table cell”. The following Python-like pseudocode tries to synthesize the basic structure of `validate_meta` nested for-loops:

```
for row_index, row in data:
    #execute checks on row
    for field, value in row.items():
        if field == 'id':
            items = value.split(' ')
            for id in items:
                #execute checks on id
        if field == 'title':
            #execute checks on 'title' value
        if field == 'author' or field == 'editor':
            items = value.split('; ')
            for resp_ag in items:
                #execute checks on resp_ag
                id_pattern = <pattern>
                ra_ids = [id for id in resp_ag if id match id_pattern]
                for id in ra_ids:
                    #execute checks on id
        if field == 'pub_date':
            #execute checks on 'pub_date' value
        if field == 'venue':
            #execute checks on 'venue' value
            id_pattern = <pattern>
            br_ids = [id for id in <'venue' value> if id match id_pattern]
            for id in br_ids:
                #execute checks on id
        if field == 'volume':
            #execute checks on 'volume' value
        if field == 'issue':
            #execute checks on 'volume' value
        if field == 'page':
            #execute checks on 'page' value
        if field == 'type':
            #execute checks on 'page' value
        if field == 'publisher':
            # execute checks on 'venue' value
            id_pattern = <pattern>
            br_ids = [id for id in <'publisher' value> if id match id_pattern]
```

```
for id in br_ids:
    # execute checks on id
```

The pseudocode above shows that, at any given state of the iteration inside a row (dictionary), the validation methods to call and whether or not to split the value into smaller (iterable) structures depend on the field that is currently being processed, i.e. on the key of the current key-value pair. On the other hand, all the rows are validated with the same methods.

Creating and storing errors

Before starting to analyze the code of the validation methods and `validate_meta` from a low-level perspective, it is useful to understand the mechanism by which the information about an error is stored in a dictionary object and added to the list returned by `validate_meta`, i.e. `error_final_report`. At every step of the iteration in the nested for-loops, `validate_meta` calls one or more methods from the validation classes (`Wellformedness`, `IdSyntax`, `IdExistence`, and `Semantics`) passing the current loop variable as input. Almost all these methods return a boolean value, except for `Wellformedness.get_missing_values` (returning a dictionary, which is empty if no error was found) and `Wellformedness.get_duplicates_meta` (which returns an empty list if no error was found, or, if the document contains duplicates, a list containing error dictionaries to add directly to `error_final_report`). The mechanism by which error dictionaries are created is the same across the whole process: if the validation method returns `True` or an empty dictionary/list, it means the input data conforms to the specific rule(s) checked by the method, and the process continues by calling other validation methods on the same piece of data or stepping to the next loop variable; otherwise, i.e. if the validation method returns `False` or the dictionary/list is not empty, the following steps take place:

1. the user message corresponding to the failed rule is retrieved from the `messages` dictionary (resulting from reading the “`messages.yaml`” file at the `Validator` class initialization) and assigned to the `message` variable
2. a tree-like dictionary representing the location of the data involved in the error within the document is assigned to the `table` variable
3. the `Helper.create_error_dict` method is called, taking as input `message`, `table`, and the values for the other parameters

4. the error dictionary returned by `create_error_dict` is appended to the final list of error dictionaries (`error_final_report`).

The following script is an example of how this procedure is implemented in `validate_meta` (the ellipsis keyword, i.e. "...", is used as a placeholder in lieu of the parts of the code that are not relevant to the example).

```
def validate_meta(self) -> list:
    error_final_report = []
    messages = self.messages
    ...
    for row_idx, row in enumerate(self.data):
        ...
        for field, value in row.items():
            ...
            if field == 'pub_date':
                if value:
                    if not self.wellformed.wellformedness_date(value):
                        message = messages['date_format']
                        table = {row_idx: {field: [0]}}
                        error_final_report.append(
                            self.helper.create_error_dict(
                                validation_level='csv_wellformedness',
                                error_type='error',
                                message=message,
                                error_label='date_format',
                                located_in='item',
                                table=table
                            )
                        )
            ...
        ...
    ...
    return error_final_report
```

Script 9. An example showing how the `create_error_dict` method is called inside the `validate_meta`. The field involved is `pub_date` of `META-CSV`.

The script shows that if the value for the key "pub_date" in the dictionary representing the row does not validate (i.e. if `wellformedness_date(value)` returns `False`), then the method `create_error_dict` is called. The arguments passed to `create_error_dict` correspond to the values of the dictionary it will return: the validation level of the error, the error type, the user message, the unique label for the error, the scope of the error, and the location of the error in the document. This last argument, `table`, is a dictionary representing the structure of the document as a tree: the `row_idx` key is the index of the currently iterated dictionary within the data list; its value is another dictionary where the key is the current field of the row (i.e. the string "pub_date"), and the value is a list containing the position of the item interested by the error within the "cell" of the table, which in this case is the first position. In this regard, it is essential to notice that this example represents the case of an error involving a field, *pub_date*, whose value contains only one item: the

string of the 'pub_date' key of a row dictionary is not split to produce a list of items (substrings of value) because no separator is defined for such field. Therefore, the first position is the only possible position for the value of 'pub_date'. In cases where the field value can contain more than one item, this string is split by its specific separator (i.e. a whitespace character (Unicode U+0020), for the *id* field and a semicolon followed by a whitespace character for the *author* and *editor* fields). The list produced by the split function is also iterated over with a for-loop, and every element in it (corresponding to each item in the field value) is passed to the call of the proper validation function. The following script shows how the value of the *id* field is processed inside `validate_meta`, illustrating which validation functions from other classes are called, where they are called, and how (i.e. what they take as input).

```
def validate_meta(self) -> list:
    error_final_report = []
    messages = self.messages
    ...
    br_id_groups = []
    for row_idx, row in enumerate(self.data):
        ...
        id_ok = True # switch for id field well-formedness
        ...
        for field, value in row.items():
            ...
            if field == 'id':
                if value:
                    br_ids_set = set() # contains only well-formed IDs
                    items = value.split(' ')
                    for item_idx, item in enumerate(items):
                        if item == '':
                            message = messages['extra_space']
                            table = {row_idx: {field: [item_idx]}}
                            error_final_report.append(
                                self.helper.create_error_dict(
                                    validation_level='csv_wellformedness',
                                    error_type='error',
                                    message=message,
                                    error_label='extra_space',
                                    located_in='item',
                                    table=table)
                            )

                        elif not self.wellformed.wellformedness_br_id(item):
                            message = messages['br_id_format']
                            table = {row_idx: {field: [item_idx]}}
                            error_final_report.append(
                                self.helper.create_error_dict(
                                    validation_level='csv_wellformedness',
                                    error_type='error',
                                    message=message,
                                    error_label='br_id_format',
                                    located_in='item',
```

```

        table=table)
    )

    else:
        if item not in br_ids_set:
            br_ids_set.add(item)
        else: # in-value duplication of the same ID
            table = {row_idx: {field: [i for i, v in
enumerate(items) if v == item]}}
            message = messages['duplicate_id']

            error_final_report.append(
                self.helper.create_error_dict(
                    validation_level='csv_wellformedness',
                    error_type='error',
                    message=message,
                    error_label='duplicate_id',
                    located_in='item',
                    table=table) # valid=False
            )

# 2nd step: SYNTAX OF ID (BIBLIOGRAPHIC RESOURCE)
if not self.syntax.check_id_syntax(item):
    message = messages['br_id_syntax']
    table = {row_idx: {field: [item_idx]}}
    error_final_report.append(
        self.helper.create_error_dict(
            validation_level='external_syntax',
            error_type='error',
            message=message,
            error_label='br_id_syntax',
            located_in='item',
            table=table)
    )

# 3rd step: EXISTENCE OF ID (BIBL. RESOURCE)
elif not self.existence.check_id_existence(item):
    message = messages['br_id_existence']
    table = {row_idx: {field: [item_idx]}}
    error_final_report.append(
        self.helper.create_error_dict(
            validation_level='existence',
            error_type='warning',
            message=message,
            error_label='br_id_existence',
            located_in='item',
            table=table, valid=True)
    )

if len(br_ids_set) >= 1: # store only well-formed IDs
    br_id_groups.append(br_ids_set)

if len(br_ids_set) != len(items): #->some error inside items
    id_ok = False
    ...
    ...
    ...
return error_final_report

```

Script 10. A reduced version of the `validate_meta` showing the sub-process to validate the value of the `id` field.

Before iterating over the rows, an empty list is initialized (`br_ids_groups`), which will be used, later in `validate_meta`, as input of `helper.group_ids`, which returns another list of sets where each set contains all the IDs that are associated with the same bibliographic entity. Then, inside the loop over the dictionaries in `data`, the `id_ok` variable is initialized as `True`: it is a “switch” whose value will be used later, in a boolean expression that is the condition to execute the semantic check on the current row (see section Semantics). Later, the loop over the keys of a dictionary starts: if the current field is `id` and its value is not empty, the variable `br_ids_set` is initialized as an empty list; inside it will be appended all the IDs that co-occur in the current field value and will have passed the checks for well-formedness, when validated individually. Then, the `items` variable is created: it is a list containing all the substrings of `value` that are formed by splitting `value` by its proper separator, i.e. a whitespace character (`items = value.split(' ')`). Then, for each element (`item`) in the `items` list, the checks of validation rules are applied sequentially:

- if the item is an empty string (`if item == ''`),⁵⁵ it means that there was an extra space in the split string, so the relevant error dictionary is created (by calling `create_error_dict` with the appropriate arguments) and added to the `final_error_report` list;
- if the above condition is not met, the format of each item is checked (`elif not self.wellformed.wellformedness_br_id(item)`), and if it fails an error is created and added to the global list;
- if both the above checks have been passed, `validate_meta` checks whether the current item is already in set of items of the current field value that have been (successfully) validated (`if item not in br_ids_set`):
 - if it is not there, it adds it to the set;
 - if it is already in the set, it means that the current item is an in-field duplicate, and the relevant error is created and stored.
- if all the previous checks have been passed (i.e. if the current item is valid for the first level of the validation, concerning the syntax of META-CSV), then the checks for the second and

⁵⁵ As concerns extra spaces, the rule is checked entirely within `validate_meta`, without calling methods from other classes.

third validation level are applied sequentially, and the relevant errors are created and added to `final_error_list`.

- the two last steps of the sub-process for the *id* item are aimed at storing the obtained results into two variables that are re-used elsewhere:
 - the set of well-formed ids, if not empty, is added to `br_id_groups`, the list of set of IDs to be processed later by `helper.group_ids` (`br_id_groups.append(br_ids_set)`);
 - the value of the `id_ok` switch is updated to `False`, if any error was found in the first validation level, making the number of elements contained in `br_ids_set` different than the number of elements in the `items` list, since only *wellformed* items have been added to `br_ids_set`.

For fields whose items have internal components (i.e. *author*, *editor*, *venue* and *publisher*, where a series of identifiers can be associated with an entity by being specified inside square brackets “within” the item itself) `validate_meta` iterates components as well, just like it does over the items containing them, with a for-loop. The only difference is that an item’s components are extracted by using a regular expression, instead of using the `split` method: a pattern is iteratively applied to the item string, and all matching substrings are put into a list, which ends up containing the identifiers associated with the entity represented by the item. The following script shows how this is done in context:

```
if field == 'author' or field == 'editor':
    if value:
        items = value.split('; ')
        for item_idx, item in enumerate(items):
            if self.wellformed.orphan_ra_id(item):
                ... #create and store error dictionary
            if not self.wellformed.wellformedness_people_item(item):
                ... #create and store error dictionary
            else:
                ids = [m.group() for m in
finditer(r'((?:crossref|orcid|viaf|wikidata|ror):\S+)(?=\s|\)]', item)]

                for id in ids:
                    #2nd step: SYNTAX OF ID (BIBLIOGRAPHIC RESOURCE)
                    if not self.syntax.check_id_syntax(id):
                        table = {row_idx: {field: [item_idx]}}
                        ... # create and store error dictionary
                    #3rd step: EXISTENCE OF ID (BIBL. RESOURCE)
                    elif not self.existence.check_id_existence(id):
                        table = {row_idx: {field: [item_idx]}}
                        ... # create and store error dictionary
```

Script 11. An excerpt from `validate_meta` showing the sub-process of `validate_meta` to validate the field values of author and editor, for which multiple items with multiple components can be specified.

After being checked for well-formedness, an item inside the value of `row['author']` or `row['editor']` is passed to the `finditer` function of the built-in `re` module.⁵⁶ This function, as stated in the documentation:⁵⁷ “return an iterator yielding match objects over all non-overlapping matches for the RE pattern in string. The string is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.” Every `Match` object in the iterator returned by `finditer` is converted to a string via the `group` method and added to a list (`ids`) with a list comprehension: `ids = [m.group() for m in finditer(r'((?:crossref|orcid|viaf|wikidata|ror):\S+)(?=\s|\s|))', item)]`. Then, for each ID in the `ids` list, the checks for the ID’s syntax and existence are executed straight away, since their well-formedness has been already checked in the previous step, which took as input the item containing the current ID. It is worth noticing that if an ID fails a check, its path in the document can be represented down to the item that contains it, but no further; in other words, even if only one component of an item is proved invalid, the whole item that contains it will result invalid, since, for an item’s component, the path is the same as the item containing it: `table = {row_idx: {field: [item_idx]}}`.

Well-formedness

The previous section has illustrated the basic structure of `validate_meta` and how methods from other classes are called inside it. The present section will first list which methods from the `Wellformedness` class are used in `validate_meta` and then delve into the technical details of one them, namely `get_missing_values`, to show, by means of an example, how the peculiar structure of the data in META-CSV requires an *ad hoc* validator. The basic principles of the validation process for the rules of the first validation step can be summarized as follows:

- the check of required fields, applied at the row level (i.e. concerning a rule involving multiple fields of the same row), is always applied on each row, regardless of the validity of the single items it contains;

⁵⁶ <https://docs.python.org/3/library/re.html>

⁵⁷ <https://docs.python.org/3/library/re.html>

- the methods for validating single items are only called if the field value containing the item is not empty, i.e. if the value of the current key-value pair in the for-loop is not an empty string;
- which methods are called for each item depend on the field of the value they are contained in, (the key of the current key-value pair in the for-loop) since each field has its own specific rules;
- checks are applied to the items of all the fields
- checks on items are executed sequentially on condition that the item has passed the previous checks, i.e. if an item does not validate for a given rule, it will not be checked for the other rules.

All the methods from the `Wellformedness` class are called inside the `validate_method` as methods of an instance of `Wellformedness`, which is assigned to the `wellformed` attribute in the initialization block of the `Validator` class. The following table lists all the methods in the `Wellformedness` class that are called inside the process of `validate_meta`, and indicates what datatype or data structure they take as input, and which fields they concern.

Method	Type/Structure	Field
<code>check_page_interval</code>	string	<i>page</i>
<code>get_duplicates_meta</code>	list	<i>id</i>
<code>get_missing_values</code>	dictionary	all conditionally required fields
<code>orphan_ra_id</code>	string	<i>author, editor, publisher</i>
<code>orphan_venue_id</code>	string	<i>venue</i>
<code>wellformedness_br_id</code>	string	<i>id, author, editor, venue, publisher</i>
<code>wellformedness_date</code>	string	<i>pub_date</i>
<code>wellformedness_page</code>	string	<i>page</i>
<code>wellformedness_type</code>	string	<i>type</i>
<code>wellformedness_venue</code>	string	<i>venue</i>
<code>wellformedness_volume_issue</code>	string	<i>volume, issue</i>

Table 4. The methods of the `Wellformedness` class and their input data.

get_missing_values

get_missing_values has the purpose of checking that each row has all the field values that need to be specified. Indeed, even if no field is always required to have a value specified for it, under certain conditions some fields cannot be empty (as shown in Figure 5); moreover, some fields, depending on the value of other fields, must have a value included in a restricted set of possible ones. get_missing_values is called in the outer for-loop of validate_meta: for each dictionary in the input list (data), the row is passed as input to get_missing_values before any other function is called, so that it is executed unconditionally, regardless of the outcome of all the other validation functions. The following script shows the context of validate_meta in which get_missing_values is called.

```
def validate_meta(self) -> list:
    error_final_report = []
    messages = self.messages
    ...
    for row_idx, row in enumerate(self.data):
        row_ok = True # switch for row well-formedness
        id_ok = True # switch for id field well-formedness
        type_ok = True # switch for type field well-formedness

        missing_required_fields = self.wellformed.get_missing_values(row)
        if missing_required_fields:
            message = messages['required_fields']
            table = {row_idx: missing_required_fields}
            error_final_report.append(
                self.helper.create_error_dict(
                    validation_level='csv_wellformedness',
                    error_type='error',
                    message=message,
                    error_label='required_fields',
                    located_in='field',
                    table=table))
            row_ok = False
        ...
    ...
    return error_final_report
```

Script 12. The sub-process of validate_meta to call get_missing_values and process its output.

First, the row_ok switch is initialized to True: which will be used as one of the conditions to call check_semantics on the row, together with the other switches, i.e. id_ok and type_ok: the value of these variables is updated to False when the rule they represent is not respected. Then a variable (missing_values) is assigned the output of get_missing_values, which – contrary to most of the validation functions – returns a dictionary. If this dictionary is empty, it means that no errors were found and the process continues validating the inner structure of the

row/dictionary; otherwise, if the `missing_keys` dictionary contains keys and values, these are directly used as the value of `table[row_idx]`, i.e. the variable storing the index of the current row in data. Indeed, the structure, keys, and set of possible values of `missing_data` are the same as the ones used to represent the path of an error inside a given row. The following script shows the code for `get_missing_values`.

```
def get_missing_values(self, row: dict) -> dict:
    missing = {}
    if not row['id']: # ID value is missing

        if row['type']: # ID is missing and 'type' is specified
            if row['type'] in ['book', 'dataset', 'data file', 'dissertation',
                              'edited book', 'journal', 'journal article', 'monograph', 'other', 'peer review',
                              'posted content', 'web content', 'proceedings article', 'reference book',
                              'report']:
                if not row['title']:
                    missing['type'] = [0]
                    missing['title'] = None
                if not row['pub_date']:
                    missing['type'] = [0]
                    missing['pub_date'] = None
                if not row['author'] and not row['editor']:
                    missing['type'] = [0]
                    if not row['author']:
                        missing['author'] = None
                    if not row['editor']:
                        missing['editor'] = None

            elif row['type'] in ['book chapter', 'book part',
                                'book section', 'book track',
                                'component', 'reference entry']:
                if not row['title']:
                    missing['type'] = [0]
                    missing['title'] = None
                if not row['venue']:
                    missing['type'] = [0]
                    missing['venue'] = None

            elif row['type'] in ['book series', 'book set', 'journal',
                                'proceedings', 'proceedings series', 'report series', 'standard', 'standard
                                series']:
                if not row['title']:
                    missing['type'] = [0]
                    missing['title'] = None

            elif row['type'] == 'journal issue':
                if not row['venue']:
                    missing['type'] = [0]
                    missing['venue'] = None
                if not row['title'] and not row['issue']:
                    missing['type'] = [0]
                    if not row['title']:
                        missing['title'] = None
                    if not row['issue']:
                        missing['issue'] = None

            elif row['type'] == 'journal volume':
                if not row['venue']:
                    missing['venue'] = None
```

```

        missing['type'] = [0]
        missing['venue'] = None
    if not row['title'] and not row['volume']:
        missing['type'] = [0]
        if not row['title']:
            missing['title'] = None
        if not row['volume']:
            missing['volume'] = None

    else: #i.e. if row['type'] is empty

        if not row['title']:
            missing['type'] = None
            missing['title'] = None
        if not row['pub_date']:
            missing['type'] = None
            missing['pub_date'] = None
        if not row['author'] and not row['editor']:
            missing['type'] = None
            if not row['author']:
                missing['author'] = None
            if not row['editor']:
                missing['editor'] = None

    if row['id']:
        if row['volume']:
            if not row['venue']:
                missing['volume'] = [0]
                missing['venue'] = None
            if row['type'] not in ['journal article', 'journal volume', 'journal
issue']:
                missing['volume'] = [0]
                if not row['type']:
                    missing['type'] = None
                else:
                    missing['type'] = [0]

        if row['issue']:
            if not row['venue']:
                missing['issue'] = [0]
                missing['venue'] = None
            if row['type'] not in ['journal article', 'journal volume', 'journal
issue']:
                missing['issue'] = [0]
                if not row['type']:
                    missing['type'] = None
                else:
                    missing['type'] = [0]

    return missing

```

Script 13. The `get_missing_values` *method of the* `Wellformedness` *class*.

`get_missing_values` general functioning is based on verifying that, for every condition upon which some field values become required, when this condition is met all the required field values are specified. If any required field value is missing, the `missing` dictionary, initialized as empty at the beginning of the process, is populated or updated in the following way:

- field values that are specified, i.e. the condition on which certain other field values are required, are represented in the dictionary through a key-value pair, where the key is the field name and the value is always `[0]` (since the fields that require the presence of other field values can never contain multiple items); e.g. `'type': [0]` would mean that the fact that the value of the *type* field is specified (or the fact that a specific value is specified for *type*) is the condition requiring that other field values be specified (or other fields take specific values);
- fields values that are missing are represented by a key-value pair where the key is the field name and the value is a list containing `None`.

Syntax of Identifiers

The syntax of each ID is checked against its identifier scheme by the `check_syntax` method of the `IdSyntax` class. In this class there is only the `check_syntax` method, and the initialization block is empty. `check_syntax` method takes as input the string corresponding to the identifier, with the prefix included. First, the prefix is extracted by slicing the string from the first character to the colon (":") character, and it is assigned to the `oc_prefix` variable. Then, a series of consecutive if-statements of the form `if oc_prefix == <ID prefix>` determines what to do next on the condition of the value of `oc_prefix`. In fact, depending on this value, the pertinent subclass of `IdManager` is instantiated and assigned to a variable: e.g. if `oc_prefix` equals to `"doi:"`, then the `DOIManager` class is instantiated. From the class instance, the `syntax_ok` method is called with the string of the ID as input. Finally, the result of `syntax_ok` is returned. If the prefix extracted by `check_syntax` from the input ID does not match any of the strings in the if-statements,⁵⁸ the method just returns `None`.

```
from oc_idmanager import doi, isbn, issn, orcid, pmcid, pmid, ror, url, viaf,
wikidata, wikipedia
```

```
class IdSyntax:
    def __init__(self):
        pass

    def check_id_syntax(self, id: str):

        oc_prefix = id[: (id.index(':') + 1)]

        if oc_prefix == 'doi:':
            vldt = doi.DOIManager()
            return vldt.syntax_ok(id)
        if oc_prefix == 'isbn:':
            vldt = isbn.ISBNManager()
```

⁵⁸ This should never happen, as the format of the string containing the ID, including the fact that each ID be prefixed correctly, has already been checked in the previous validation step.

```

        return vldt.syntax_ok(id)
    if oc_prefix == 'issn:':
        vldt = issn.ISSNManager()
        return vldt.syntax_ok(id)
    if oc_prefix == 'orcid:':
        vldt = orcid.ORCIDManager()
        return vldt.syntax_ok(id)
    if oc_prefix == 'pmcid:':
        vldt = pmcid.PMCIDManager()
        return vldt.syntax_ok(id)
    if oc_prefix == 'pmid:':
        vldt = pmid.PMIDManager()
        return vldt.syntax_ok(id)
    if oc_prefix == 'ror:':
        vldt = ror.RORManager()
        return vldt.syntax_ok(id)
    if oc_prefix == 'url:':
        vldt = url.URLManager()
        return vldt.syntax_ok(id)
    if oc_prefix == 'viaf:':
        vldt = viaf.VIAFManager()
        return vldt.syntax_ok(id)
    if oc_prefix == 'wikidata:':
        vldt = wikidata.WikidataManager()
        return vldt.syntax_ok(id)
    if oc_prefix == 'wikipedia:':
        vldt = wikipedia.WikipediaManager()
        return vldt.syntax_ok(id)

```

Script 14. The IdSyntax class initializer and its check_id_syntax method, from which the methods from oc-idmanager are called to validate an ID according to its syntax.

The `check_syntax` method is called by `validate_meta` for each item or item component of identifier type, always on condition that it has passed the previous checks at the first step of the validation (the ones executed with the pertinent methods of the `Wellformedness` class).

Existence of Identifiers

The existence of an ID is verified by calling from `validate_meta` the only method of the `IdExistence` class, the `check_existence` method, and passing to it the ID string as input.

The process to verify the existence of the ID is extremely similar to the one for validating its syntax, with the only difference that, while `check_syntax` makes use of the `syntax_ok` method of the classes imported from `oc_idmanager`, `check_existence` uses their `exists` method, instead. Moreover, since `exists` takes as input the ID string *without* its prefix, this is removed after it has been used to decide which class must be instantiated.

```

from oc_idmanager import doi, isbn, issn, orcid, pmcid, pmid, ror, url, viaf,
wikidata, wikipedia

class IdExistence:
    def __init__(self):
        pass

```

```

def check_id_existence(self, id:str):
    oc_prefix = id[: (id.index(':')+1)]

    if oc_prefix == 'doi:':
        vldt = doi.DOIManager()
        return vldt.exists(id.replace(oc_prefix, '', 1))
    if oc_prefix == 'isbn:':
        vldt = isbn.ISBNManager()
        return vldt.exists(id.replace(oc_prefix, '', 1))
    if oc_prefix == 'issn:':
        vldt = issn.ISSNManager()
        return vldt.exists(id.replace(oc_prefix, '', 1))
    if oc_prefix == 'orcid:':
        vldt = orcid.ORCIDManager()
        return vldt.exists(id.replace(oc_prefix, '', 1))
    if oc_prefix == 'pmcid:':
        vldt = pmcid.PMCIDManager()
        return vldt.exists(id.replace(oc_prefix, '', 1))
    if oc_prefix == 'pmid:':
        vldt = pmid.PMIDManager()
        return vldt.exists(id.replace(oc_prefix, '', 1))
    if oc_prefix == 'ror:':
        vldt = ror.RORManager()
        return vldt.exists(id.replace(oc_prefix, '', 1))
    if oc_prefix == 'url:':
        vldt = url.URLManager()
        return vldt.exists(id.replace(oc_prefix, '', 1))
    if oc_prefix == 'viaf:':
        vldt = viaf.VIAFManager()
        return vldt.exists(id.replace(oc_prefix, '', 1))
    if oc_prefix == 'wikidata:':
        vldt = wikidata.WikidataManager()
        return vldt.exists(id.replace(oc_prefix, '', 1))
    if oc_prefix == 'wikipedia:':
        vldt = wikipedia.WikipediaManager()
        return vldt.exists(id.replace(oc_prefix, '', 1))

```

Script 15. The IdExistence class initializer and its check_id_existence method, from which the methods from oc-idmanager are called to verify the existence of an ID.

The `check_existence` method is called by `validate_meta` for each item or item component of identifier type, always on condition that it has passed the previous checks at the first step of the validation (the ones executed with the pertinent methods of the `Wellformedness` class), as well as the checks at the second step of the validation (the ones executed with `check_syntax`).

Semantics

The semantics expressed in the row are validated with a function from the `Semantics` class, named `check_semantics`. It is called from `validate_meta` as the last operation to perform on the current row of the loop, only if the row, the items inside the *id* field, and the item inside the *type* field have been all proven valid for the first validation level, i.e. when they are all wellformed. The script below shows that an if-statement verifies that the boolean expression “`row_ok and id_ok and`

`type_ok`” equals to `True`; the variables involved in it store the boolean values corresponding to the outcome of validating the row dictionary, the ID strings in the `items` list for `row['id']`, and the string for `row['type']`. If the condition is met, the `invalid_semantics` variable is assigned the output of `check_semantics`, which returns an empty dictionary if the semantics expressed in the row is valid, and otherwise, i.e. if it is invalid, it returns a dictionary specifying the position of the elements involved in the error. If the dictionary returned by `check_semantics` contains anything (if `invalid_semantics`), i.e. if the row contains some error concerning semantics, the appropriate message is retrieved from the `messages` dictionary, and the variable `table` is directly assigned `invalid_semantics` (due to the fact that the output of `check_semantics` is built following the model of the table dictionary of `create_error_dict`).

```
if row_ok and id_ok and type_ok:
    invalid_semantics = self.semantics.check_semantics(row, id_type_dict)
    if invalid_semantics:
        message = messages['row_semantics']
        table = {row_idx: invalid_semantics}
        error_final_report.append(self.helper.create_error_dict(
            validation_level='semantics',
            error_type='error',
            message=message,
            error_label='row_semantics',
            located_in='field',
            table=table))
```

Script 16. *An excerpt of `validate_meta` showing how the `check_semantics` method is called (only when the row, the IDs and the type are well-formed).*

The `check_semantics` method takes as input the dictionary representing the row and the dictionary embodying the ID-type alignment, i.e. the definition of what identifier types can be associated with a bibliographic resource of a given type. This dictionary (the variable named `id_type_dict` in `validate_meta`) is one of the instance attributes of the `Validator` class and derives from reading and converting to a dictionary the “`id_type_alignment.json`” JSON file.

```
def check_semantics(self, row: dict, alignment: dict) -> dict:

    invalid_row = {}
    row_type = row['type']
    row_ids = row['id'].split(' ') # list
    invalid_ids_idxes = []

    for id_idx, id in enumerate(row_ids):
        if id[:id.index(':')] not in alignment[row_type]:
            invalid_ids_idxes.append(id_idx)
    if invalid_ids_idxes:
        invalid_row['id'] = invalid_ids_idxes
```

```

        invalid_row['type'] = [0]
    return invalid_row

```

Script 17. The `check_semantics` method of the `Semantics` class.

The method first creates an empty dictionary, `invalid_row`, that will store the path of the row's elements that are involved in any error that is found. Next, it retrieves the value of the 'type' key from the `row` dictionary, and assigns it to the `row_type` variable. It then retrieves the value of the 'id' key in `row`, splits the string into a list of individual IDs, and stores this list in the `row_ids` variable. The method then creates an empty list called `invalid_ids_idx`s that will store the indexes of any IDs that are found incompatible with the value of `row_type`. For each ID in the `row_ids` list, the method checks if it is among the valid IDs for the bibliographic resource's type; it does so by taking the ID's abbreviation (the characters of the ID's prefix before the colon) and seeing if it is in the list associated with the current type in the `alignment` dictionary. If the abbreviation is not found, the index of the ID causing the invalidity is added to the `invalid_ids_idx`s list. If any invalid IDs were found in the row, the method adds the index(es) of the invalid ID(s) to the `invalid_row` dictionary under the key "id". It also sets the value of the "type" key to a list containing the integer of the position of the only item in the type field, i.e. [0]. Finally, the method returns the `invalid_row` dictionary, which will be empty if no invalid IDs were found in the row.

Validating CITS-CSV

The process for validating a CITS-CSV document is executed by the `validate_cits` method of the `Validator` class. At its core, `validate_cits`'s structure is built on the same principle as `validating_meta`: for each row, represented as a dictionary in `data`, the methods taking `row` as input are executed; for each key-value pair (field-field value pair) in `row`, depending on the current key, a validation method is executed taking as input either the whole string of the value (`citing_publication_date` and `cited_publication_date`) or every element in the list deriving from splitting it (`citing_id` and `cited_id`). However, compared to `validate_meta`, `validate_cits` is much less complex, due to the lesser number of key-value pairs in the dictionaries in `data`, and to the fact that each row can only contain two types of data, i.e. identifiers and dates. Indeed, the validation methods called by `validate_cits` from the other classes are only five: `wellformedness_br_id` (which checks that each item in the 'citing_id' and 'cited_id' values is formatted correctly), `check_id_syntax`, `check_id_existence`, `wellformedness_date` (checking that the value of `row[citing_publication_date]` or `row[cited_publication_date]` is formatted

correctly) and `get_duplicates_cits` (which looks if there is any duplicate citation in the table). As for managing the errors, it follows the same mechanism of `validate_meta`: whenever an error is found at any given state of the iteration, `create_error_dict` is called with the arguments pertaining the error and its location in the document. The following script illustrates the body of `validate_cits`.

```
def validate_cits(self) -> list:
    error_final_report = []

    messages = full_load(open('validator/messages.yaml', 'r', encoding='utf-8'))

    id_fields_instances = []

    for row_idx, row in enumerate(self.data):
        for field, value in row.items():
            if field == 'citing_id' or field == 'cited_id':
                if not value: # Check required fields
                    message = messages['required_value_cits']
                    table = {row_idx: {field: None}}
                    error_final_report.append(
                        self.helper.create_error_dict(
                            validation_level='csv_wellformedness',
                            error_type='error',
                            message=message,
                            error_label='required_value_cits',
                            located_in='field',
                            table=table))
            else: # i.e. if string is not empty...
                ids_set = set() # set where to put valid IDs only
                items = value.split(' ')

                for item_idx, item in enumerate(items):

                    if item == '':
                        message = messages['extra_space']
                        table = {row_idx: {field: [item_idx]}}
                        error_final_report.append(
                            self.helper.create_error_dict(
                                validation_level='csv_wellformedness',
                                error_type='error',
                                message=message,
                                error_label='extra_space',
                                located_in='item',
                                table=table))

                    elif not self.wellformed.wellformedness_br_id(item):
                        message = messages['br_id_format']
                        table = {row_idx: {field: [item_idx]}}
                        error_final_report.append(
                            self.helper.create_error_dict(
                                validation_level='csv_wellformedness',
                                error_type='error',
                                message=message,
```

```

        error_label='br_id_format',
        located_in='item',
        table=table))

    else:
        if item not in ids_set:
            ids_set.add(item)
        else: # in-field duplication of the same ID
            table = {row_idx: {field: [i for i, v in
enumerate(items) if v == item]}}
            message = messages['duplicate_id']

            error_final_report.append(
                self.helper.create_error_dict(
                    validation_level='csv_wellformedness',
                    error_type='error',
                    message=message,
                    error_label='duplicate_id',
                    located_in='item',
                    table=table))

            # 2nd step: EXTERNAL SYNTAX OF ID (bibl. res.)
            if not self.syntax.check_id_syntax(item):
                message = messages['br_id_syntax']
                table = {row_idx: {field: [item_idx]}}
                error_final_report.append(
                    self.helper.create_error_dict(
                        validation_level='external_syntax',
                        error_type='error',
                        message=message,
                        error_label='br_id_syntax',
                        located_in='item',
                        table=table))

            # 3rd step: EXISTENCE OF ID (bibl. res.)
            elif not self.existence.check_id_existence(item):
                message = messages['br_id_existence']
                table = {row_idx: {field: [item_idx]}}
                error_final_report.append(
                    self.helper.create_error_dict(
                        validation_level='existence',
                        error_type='warning',
                        message=message,
                        error_label='br_id_existence',
                        located_in='item',
                        table=table,
                        valid=True))

        if len(ids_set) >= 1:
            id_fields_instances.append(ids_set)

        if field == 'citing_publication_date' or field ==
'cited_publication_date':
            if value:
                if not self.wellformed.wellformedness_date(value):
                    message = messages['date_format']

```

```

        table = {row_idx: {field: [0]}}
        error_final_report.append(
            self.helper.create_error_dict(
                validation_level='csv_wellformedness',
                error_type='error',
                message=message,
                error_label='date_format',
                located_in='item',
                table=table))

    # GET BIBLIOGRAPHIC ENTITIES
    entities = self.helper.group_ids(id_fields_instances)
    # GET SELF-CITATIONS AND DUPLICATE CITATIONS (returns the list of error
reports)
    duplicate_report = self.wellformed.get_duplicates_cits(entities=entities,
                                                            data_dict=self.data,
                                                            messages=messages)

    if duplicate_report:
        error_final_report.extend(duplicate_report)

    # write error_final_report to external JSON file
    with open(join(self.output_dir, 'out_validate_cits.json'), 'w',
encoding='utf-8') as f:
        dump(error_final_report, f)

    # write human-readable validation summary to txt file
    textual_report = self.helper.create_validation_summary(error_final_report)
    with open(join(self.output_dir, "cits_validation_summary.txt"), "w",
encoding='utf-8') as f:
        f.write(textual_report)

    return error_final_report

```

Script 18. The validate_cits method.

Software evaluation

This chapter presents an evaluation of the software from a quantitative point of view, together with the methodology followed to perform it and the obtained results.

The approach chosen to evaluate the software consists of executing the process with diverse input documents a number of consecutive times, so as to be able to build a representative sample of the possible execution conditions. For each execution, the start time and end time of the process are used to calculate the execution time; this is then used to get metrics relative to the whole sample (mean and standard deviation, median, minimum time and maximum time of execution) and so get reliable indicators of the software performance in terms of execution time.

To ensure the reproducibility of the evaluation test, the significant details about the hardware and software that have been employed are listed below:

- RAM: 12,0 GB (of which 11,9 GB usable)
- CPU: Intel(R) Core(TM) i7-8565U @ 1.80GHz
- Operating system: Windows 11 (64bit)
- Internet connection (WiFi 5):
 - Bandwidth: 2.4 GHz
 - Transmission speed: up to ~ 170 Mbps

The performance was measured by running the software 10 consecutive times on four documents: two instances of CITS-CSV and two instances of META-CSV. For each table type, one document is completely valid and the other is invalid, containing errors that have been manually inserted in a copy of the valid document. Both the invalid document of CITS-CSV type and the invalid document of META-CSV type contain at least one instance of all the error kinds that are applicable to either table type. This means that in the error report (list of error dictionaries) of the invalid documents, 9 and 19 different values for the `error_label` key are present for the CITS-CSV and the META-CSV document respectively. All four documents contain 15 rows. For collecting the sample data, i.e. the time taken by each of the 10 executions, and calculating the metrics for the evaluation (median, mean and standard deviation, minimum time and maximum time) the following Python script was employed:

```
from validator.main import Validator
```

```

import time
import statistics

v = Validator(csv_doc, output_dir)
execution_times = [] # list to store the execution times

for i in range(10):
    start_time = time.perf_counter() # start the timer
    v.validate() # run process
    end_time = time.perf_counter() # stop the timer
    execution_time = end_time - start_time
    execution_times.append(execution_time)

min= min(execution_times)
max = max(execution_times)
median = statistics.median(execution_times)
mean = statistics.mean(execution_times)
stand_dev= statistics.stdev(execution_times)

```

Script 19. The script used to perform the software evaluation.

	Invalid CITS-CSV	Valid CITS-CSV	Invalid META-CSV	Valid META-CSV
Median (seconds)	6.3076	5.1090	8.2806	7.7133
Mean (seconds)	6.2612	5.3930	8.2090	8.6483
Stand. dev. (seconds)	0.5943	1.1428	0.5476	2.5270
Min. time (seconds)	5.4487	4.0808	7.1232	6.3522
Max. time (seconds)	7.3080	7.9856	8.9952	14.5292
File size (kB)	1.174	1.076	4.255	4.133

Table 5. The results of running the process on small-sized valid and invalid input 10 consecutive times (each table contains 15 rows).

The results of the computation, as can be seen in Table 5, show that for the CITS-CSV tables, the average execution time of the process is slightly higher when the document contains errors: for the processing of the invalid document the mean of the values of the 10 executions is around 6.26 seconds, with a standard deviation of 0.59 seconds; for the processing of the valid document, the mean time is 5.39 seconds, with a standard deviation of 1.14 seconds. On the other hand, this relation is inverted for the execution time of the process for META-CSV tables, as the valid document, on average, takes slightly more than the valid document to be validated: the mean time for processing the invalid document is 8.21 seconds, with a standard deviation of 0.55 seconds, while the mean time for the valid document is 8.65 seconds, with a standard deviation of 2.53 seconds.

Among the factors that might cause the processing of a valid document to be more time-consuming than processing a valid one there is the fact that, if the document does not contain errors in a piece of data, then *all* the internal sub-processes are executed for that data; on the contrary, the errors contained in invalid documents prevent a single piece of data that has failed a validation check to undergo the subsequent checks, therefore saving time overall. However, it should also be considered that processing an invalid document means that every time an error is found, i.e. every time a validation function gets executed and returns False (or, in some cases, a dictionary), the function for creating and storing the error dictionary is executed, while it is never executed if the input document is completely valid. Besides the number of errors in the document, their quality might certainly be impactful, too, on the execution time of the process. In particular, the errors of the first and second validation levels that involve an ID, by blocking the execution of the third level for that ID, prevent the program to access the API service, which is a time-consuming operation. To observe the impact of API calls, the same evaluation metrics listed in the table above have been applied to two other tables, both of META-CSV type, completely valid, and containing 500 rows each. Actually, these two tables are two different versions of the same table: one version contains, for every row, exactly one DOI identifier on the *id* field (no other identifiers are present in the whole row); the other version is produced by manually removing all the identifiers from the first version, and leaving the rest of the content as it is. This is done to try to quantify the impact of the use of APIs: since the program always requests (syntactically correct) DOIs to the dedicated service, by having one valid DOI per row we make sure that 500 requests are made for the document containing IDs, while no request is made for the document that does not contain any ID. At the same time – by keeping the number of identifiers limited to one per row – we minimize the difference in size between the two versions, and consequently the difference in the number of (ID-related) operations that are executed for the two versions of the document.

	META-CSV with IDs	META-CSV with no IDs
Median (seconds)	184.6189	57.9104
Mean (seconds)	185.5193	63.8257
Stand. dev. (seconds)	30.9976	12.1845
Min. time (seconds)	150.2678	53.5255
Max. time (seconds)	266.5097	87.9281
File size (kB)	125.623	111.471

Table 6. The results of running the process on large (500 rows) valid META-CSV tables 10 consecutive times. The values under the column “META-CSV with IDs” refer to the table containing identifiers, the values under the column “META-CSV with no IDs” refer to the table from which the identifiers have been manually removed.

As can be observed from Table 6, there is a significant difference in the process execution time depending on which version of the table was passed as input: looking at the mean execution time, it can be seen that processing the document containing IDs, i.e. requiring to execute API calls, took on average 185.52 seconds; on the other hand, processing the document without IDs has a mean execution time of 63.83 seconds, being approximately three times faster. A further indicator of the impact that API requests have on the overall process is the value of the standard deviation, which is much higher for the executions involving IDs. Indeed, when the process relies on an Internet connection, the execution time might be affected by other factors besides the implementational features of the software, such as the stability of the connection itself.

Conclusions and future developments

Conclusions

The aim of this thesis was to present a methodology and a software for the validation of citation data and bibliographic metadata according to the OCDM, OpenCitations Data Model (Daquino, Peroni, and Shotton 2020; Daquino et al. 2020). The definition of the methodology and the software implementation have been conducted taking into consideration two specific use cases, namely the validation of tabular data formatted as CITS-CSV tables and META-CSV tables (Massari 2022; Massari and Heibi 2022). These two formats are used in the OpenCitations infrastructure to model input data of two different processes: the process for populating the OpenCitations Indexes, containing citations, and the process for populating OpenCitations Meta (Massari 2022b), containing bibliographic metadata. Moreover, CITS-CSV and META-CSV can be used by users to submit their data to OpenCitations, in order to be ingested and published in CROCI (Heibi, Peroni, and Shotton 2019a), the OpenCitations index of crowdsourced citations, and OpenCitations Meta.

A review of the existing schema languages and software for data validation (presented in the first chapter) has revealed the necessity of a new, *ad hoc* tool for the validation of citation data and bibliographic metadata. In fact, the specificity and the complexity of the requirements imposed by the OCDM make the existing solutions unsuitable for the task of validating the tabular documents in question.

For this reason, as explained in the second chapter, a new methodological approach for the validation of this type of data has been defined, with particular regard to the structure of the validation process and the nature of its output.

The output of the validation process has been designed to be as informative as possible about the validity of the document. Accordingly, the output of every execution of the process makes it possible to get and understand all the errors that can be found in the whole document, rather than just stating whether the document is valid or not. Each execution of the process examines the full document (and does not stop on the first error found) to ensure that the output contains all the errors, therefore being informative with regard to their number. The generated output provides a way to get the exact location of each error within the document, i.e. explicitly stating which pieces of data are causing the error. In addition, the errors in the output are categorized and uniquely labeled, allowing machines to understand the nature of the errors and process them accordingly. At the same time, a natural language explanation allows human agents to interpret them and intervene accordingly. Indeed, the

combination of machine readability and human readability is a paramount feature of the output of the validation process.

For each rule contributing to defining the validity of the document, there is a software functionality aimed at checking that it is respected. As a means to guide the implementation of the software, a categorization of these functionalities has been devised. In fact, the steps of the validation process (and consequently the rules they verify) are grouped into four “validation levels”, which are applied sequentially on the document. The first validation level concerns the syntax of the specific table type (CITS-CSV or META-CSV); the second level concerns the validation of the public IDs against their specific syntax (i.e. their identifier scheme); the third level concerns verifying that these IDs are actually publicly registered as such (i.e. that they exist); and the fourth level concerns the validation of the data with regard to the semantics it expresses (particularly, the compatibility between the type of a bibliographic resource and the identifiers attributed to it). Particular attention has been devoted to the validation of identifiers: for this task, a pre-existing piece of software by OpenCitations, *oc-idmanager*, was re-used and extended.

The methodology has been implemented in a piece of software written in Python (described at a high level in the third chapter and technically in the fourth chapter). The software was structured in classes, in order to grant easier maintainability, extensibility, and adaptability to different needs.

The software has been evaluated (as written in the fifth chapter) with respect to the execution time of the process. When executed on a 15 rows-long table, the process takes an average time of approximately 6 seconds for a CITS-CSV document, and approximately 8 seconds for META-CSV.

Future developments

The work presented in this thesis opens to further developments and enhancements, including extending it to get a wider coverage of the aspects to validate, improving the software usability, and integrating it into a data submission workflow.

The implemented software might be extended by integrating into it the possibility of validating semantic aspects of data with another tool, the Python library *pySHACL*, which is described in the first chapter. *pySHACL* is an implementation of the SHACL language, which allows to validate RDF graphs by defining a set of shapes (i.e. schemas) they must conform to. SHACL-based validation would permit to perform further checks on the semantics of the data, by directly verifying that the conversion of the tabular data into RDF format outputs data that is compliant with the OCDM.

While the validation with pySHACL would be mostly applicable for internal use in OpenCitations, other enhancements regard the fruition of the validation software from the perspective of external users. In particular, a user interface is required, to allow anyone interested in submitting data to the organization to understand the validation output more easily. The interface should provide the user an interactive workflow for the submission and correction of the document. A graphical interface, for example, might be leveraged to help the user in the localization, interpretation, and correction of possible mistakes, by highlighting critical table positions and explicitly stating, in context, the reasons for any invalidity.

In addition to the aforementioned points, the validation software might be integrated into a workflow for data submission based on GitHub issues, described in the README.md file of the “crowdsourcing” repository by OpenCitations.⁵⁹ In this workflow, the user is meant to upload data by opening GitHub issues on the dedicated repository; this data would be either accepted or rejected depending on the outcome of the validation performed by the software described in this thesis.

⁵⁹ <https://github.com/opencitations/crowdsourcing>

Bibliography

- Biron, Paul V., and Ashok Malhotra. 2004. 'XML Schema Part 2: Datatypes Second Edition'. W3C recommendation. W3C. <https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/>.
- Bray, Tim, Jean Paoli, C.M. Sperger-McQueen, Eve Maler, and François Yergeau. 2008. 'Extensible Markup Language (XML) 1.0 (Fifth Edition)'. W3C Recommendation. World Wide Web Consortium. <https://www.w3.org/TR/xml/>.
- Clark, James, and Makoto Murata. 2001. 'RELAX NG Specification'. OASIS Committee Specification. OASIS. <https://www.oasis-open.org/committees/relax-ng/spec-20011203.html>.
- Daquino, Marilena, Silvio Peroni, and David Shotton. 2020. 'The OpenCitations Data Model'. figshare. <https://doi.org/10.6084/M9.FIGSHARE.3443876>.
- Daquino, Marilena, Silvio Peroni, David Shotton, Giovanni Colavizza, Behnam Ghavimi, Anne Lauscher, Philipp Mayr, Matteo Romanello, and Philipp Zumstein. 2020. 'The OpenCitations Data Model'. In *The Semantic Web – ISWC 2020*, edited by Jeff Z. Pan, Valentina Tamma, Claudia d'Amato, Krzysztof Janowicz, Bo Fu, Axel Polleres, Oshani Seneviratne, and Lalana Kagal, 447–63. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-62466-8_28.
- Fallside, David, and Priscilla Walmsley. 2004. 'XML Schema Part 0: Primer Second Edition'. W3C recommendation. W3C. <https://www.w3.org/TR/2004/REC-xmlschema-0-20041028/>.
- Heibi, Ivan, Silvio Peroni, and David Shotton. 2019a. 'Crowdsourcing Open Citations with CROCI -- An Analysis of the Current Status of Open Citations, and a Proposal'. arXiv. <https://doi.org/10.48550/arXiv.1902.02534>.
- . 2019b. 'Software Review: COCI, the OpenCitations Index of Crossref Open DOI-to-DOI Citations'. *Scientometrics* 121 (2): 1213–28. <https://doi.org/10.1007/s11192-019-03217-6>.
- Kilpeläinen, Pekka, and Derick Wood. 1997. 'SGML and Exceptions'. In *Principles of Document Processing*, edited by Charles Nicholas and Derick Wood, 39–49. Lecture Notes

in Computer Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/3-540-63620-X_54.

———. 2001. ‘SGML and XML Document Grammars and Exceptions’. *Information and Computation* 169 (2): 230–51. <https://doi.org/10.1006/inco.2000.2964>.

Kontokostas, Dimitris, and Holger Knublauch. 2017. ‘Shapes Constraint Language (SHACL)’. W3C recommendation. W3C.

Labra Gayo, Jose Emilio, Eric Prudhommeaux, Iovka Boneva, and Dimitris Kontokostas. 2017. *Validating RDF Data*. Vol. 7. Synthesis Lectures on the Semantic Web: Theory and Technology 1. Morgan & Claypool Publishers LLC. <https://doi.org/10.2200/s00786ed1v01y201707wbe016>.

Lee, Dongwon, and Wesley W. Chu. 2000. ‘Comparative Analysis of Six XML Schema Languages’. *ACM SIGMOD Record* 29 (3): 76–87. <https://doi.org/10.1145/362084.362140>.

Martín-Martín, Alberto. 2021. ‘Coverage of Open Citation Data Approaches Parity with Web of Science and Scopus’. Billet. *OpenCitations Blog* (blog). 27 October 2021. <https://opencitations.hypotheses.org/1420>.

Martín-Martín, Alberto, Mike Thelwall, Enrique Orduna-Malea, and Emilio Delgado López-Cózar. 2021. ‘Google Scholar, Microsoft Academic, Scopus, Dimensions, Web of Science, and OpenCitations’ COCI: A Multidisciplinary Comparison of Coverage via Citations’. *Scientometrics* 126 (1): 871–906. <https://doi.org/10.1007/s11192-020-03690-4>.

Massari, Arcangelo. 2022a. ‘How to Produce Well-Formed CSV Files for OpenCitations’, May. <https://doi.org/10.5281/zenodo.6597141>.

———. 2022b. ‘A Technical Overview of OpenCitations Meta’. Billet. *OpenCitations Blog* (blog). 20 December 2022. <https://opencitations.hypotheses.org/3140>.

Massari, Arcangelo, and Ivan Heibi. 2022. ‘How to Structure Citations Data and Bibliographic Metadata in the OpenCitations Accepted Format’. arXiv. <https://doi.org/10.48550/arXiv.2206.03971>.

Mendelsohn, Noah, David Beech, Murray Maloney, and Henry Thompson. 2004. ‘XML Schema Part 1: Structures Second Edition’. W3C recommendation. W3C. <https://www.w3.org/TR/2004/REC-xmlschema-1-20041028/>.

Nüst, Daniel, Gazi Yücel, Anette Cordts, and Christian Hauschke. 2023. ‘Enriching the Scholarly Metadata Commons with Citation Metadata and Spatio-Temporal Metadata to Support Responsible Research Assessment and Research Discovery’.
<https://doi.org/10.48550/ARXIV.2301.01502>.

Pareti, Paolo, and George Konstantinidis. 2021. ‘A Review of SHACL: From Data Validation to Schema Reasoning for RDF Graphs’. arXiv.
<https://doi.org/10.48550/arXiv.2112.01441>.

Peroni, Silvio, and David Shotton. 2018a. ‘Open Citation: Definition’, 95436 Bytes.
<https://doi.org/10.6084/M9.FIGSHARE.6683855>.

———. 2018b. ‘The SPAR Ontologies’. In *The Semantic Web – ISWC 2018*, edited by Denny Vrandečić, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, 119–36. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-00668-6_8.

———. 2020. ‘OpenCitations, an Infrastructure Organization for Open Scholarship’. *Quantitative Science Studies* 1 (1): 428–44. https://doi.org/10.1162/qss_a_00023.

Persiani, Simone, Marilena Daquino, and Silvio Peroni. 2022. ‘A Programming Interface for Creating Data According to the SPAR Ontologies and the OpenCitations Data Model’. In *The Semantic Web*, edited by Paul Groth, Maria-Esther Vidal, Fabian Suchanek, Pedro Szekley, Pavan Kapanipathi, Catia Pesquita, Hala Skaf-Molli, and Minna Tamper, 305–22. Lecture Notes in Computer Science. Cham: Springer International Publishing.
https://doi.org/10.1007/978-3-031-06981-9_18.

Pezoa, Felipe, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. ‘Foundations of JSON Schema’. In *Proceedings of the 25th International Conference on World Wide Web*, 263–73. WWW ’16. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee.
<https://doi.org/10.1145/2872427.2883029>.

Prud’hommeaux, Eric, Iovka Boneva, Jose Emilio Labra Gayo, and Gregg Kellogg. 2019. ‘Shape Expressions Language 2.1’. W3C Final Community Group Report. W3C Shape Expressions Community Group. <http://shex.io/shex-semantics-20191008/>.

- Prud'hommeaux, Eric, Jose Emilio Labra Gayo, and Harold Solbrig. 2014. 'Shape Expressions: An RDF Validation and Transformation Language'. In *Proceedings of the 10th International Conference on Semantic Systems*, 32–40. SEM '14. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2660517.2660523>.
- Ray, Erik T., and Christopher R. Maden. 2003. *Learning XML, 2nd Edition*. O'Reilly.
- Robie, Jonathan, Scott Boag, Don Chamberlin, Jerome Simeon, Mary Fernandez, Michael Kay, and Anders Berglund. 2010. 'XML Path Language (XPath) 2.0 (Second Edition)'. W3C recommendation. W3C. <https://www.w3.org/TR/2010/REC-xpath20-20101214/>.
- Sahoo, Satya, Deborah McGuinness, and Timothy Lebo. 2013. 'PROV-O: The PROV Ontology'. W3C recommendation. W3C. <https://www.w3.org/TR/prov-o/>.
- Sanderson, Robert, Paolo Ciccarese, and Herbert Van de Sompel. 2013. 'Designing the W3C Open Annotation Data Model'. In *Proceedings of the 5th Annual ACM Web Science Conference*, 366–75. WebSci '13. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2464464.2464474>.
- Shotton, David. 2013. 'Publishing: Open Citations'. *Nature* 502 (7471): 295–97. <https://doi.org/10.1038/502295a>.
- . 2018. 'Citations as First-Class Data Entities: Citation Descriptions'. Billet. *OpenCitations Blog* (blog). 22 February 2018. <https://opencitations.hypotheses.org/820>.
- Vlist, Eric van der. 2002. *XML Schema*. O'Reilly. <https://www.semanticscholar.org/paper/XML-Schema-Vlist/cef48d4afd7982b6ce3623b1097c45d20c6a7a8b>.
- Walmsley, P. 2012. *Definitive XML Schema, 2nd Edition*. Prentice Hall. <https://www.semanticscholar.org/paper/Definitive-XML-Schema%2C-2nd-Edition-Walmsley/6c83eb322c14b7e0bf25cb81628dfde60dab230c>.
- Wright, Austin, Henry Andrews, and Ben Hutton. 2022. 'JSON Schema Validation: A Vocabulary for Structural Validation of JSON'. Internet Draft draft-bhutton-json-schema-validation-01. Internet Engineering Task Force. <https://datatracker.ietf.org/doc/draft-bhutton-json-schema-validation-01>.
- Wright, Austin, Henry Andrews, Ben Hutton, and Greg Dennis. 2022. 'JSON Schema: A Media Type for Describing JSON Documents'. Internet Draft draft-bhutton-json-schema-01.

Internet Engineering Task Force. <https://datatracker.ietf.org/doc/draft-bhutton-json-schema-01>.