Conception and realisation of an automatic bibliographic metadata update handler based on patch extraction and merging for the CERN document repository environment.

Master's Thesis

Martin Vesper
778805

March 4, 2015

Thesis supervisors:
Prof. Dr. Peer Ueberholz
Wojciech Ziolek

Statutory Declaration

I declare that I have developed and written the enclosed Master's Thesis completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. The Master's Thesis was not used in the same or in a similar version to achieve an academic grading or is being published elsewhere.

Abstract

Scientific literature and its corresponding bibliographic metadata information is typically available through online digital repositories:

- INSPIRE, the High Energy Physics (HEP) information system is the source of information about the whole HEP literature.
- The CERN Document Server (CDS) is the CERN Institutional Library containing all documents produced at CERN;
- arXiv is a pre-print server hosting pre-print versions of several scientific fields.
- SCOAP3 is an initiative to convert key journals in the HEP field to open access and comes with its own digital repository.

Across these 4 entities, there is a big overlap in terms of content, and maintaining consistency between the corresponding bibliographic metadata is an open challenge.

The proposed thesis tries to model and implement a possible solution to automate the propagation of updates in order to reduce the necessary manual data manipulation to a minimum.

Contents

1 Introduction

Publishing a scientific article is a lengthy process that in general does not start with submitting a perfectly curated version to one of the many well known publishers. In recent days it often starts with early published preprint versions, which are accessible by online repositories [1].

Depending on the importance of those articles, curators[2] of those repositories will start the curation process on those early versions, which leads to the following problem: The curators might make manual adjustments to certain fields of the metadata, like fixing typos or normalizing author names for the system, which will conflict with future iterations of the article in the publishing process.

Resolving those conflicts requires, in general, an operator (some kind of system administrator for example) picking the changes applied by the publisher while maintaining the ones done by the curators. This process can get elaborate considering multiple intertwined repositories with different metadata requirements.

This thesis proposes a solution to this problem by a configurable, automated merging tool that can be utilized in the environment of CERN, as the initiator of this thesis, and the four repositories SCOAP3, INSPIRE, CERN Document Server and arXiv, which are described in sections below.

1.1 CERN

CERN, the European Organization for Nuclear Research, is a research center located in Geneva that explores the fundamental structure of the universe [2] by utilizing the worlds largest and most powerful particle accelerator , the 'Large Hadron Collider' (LHC). This 27 kilometer long ring of superconducting magnets is equipped with multiple different experiments [3], for example the 'A Toroidal LHC Apparatus' (ATLAS) [4] and 'Compact Muon Solenoid' (CMS) [5] experiment.

The research in the field of High Energy Phisics (HEP) at CERN is state of the art, theoretically and practically. Therefore it is not surprising that researcher at CERN have a high output of papers (1594 publications in 2013[6]) and articles, which in turn leads to the previously described problem of manual metadata merges.

[1] a repository in general refers to a storage location. In the case of this thesis, it means a storage for digital objects.

[2] "A keeper or custodian of a museum or other collection"[1]

1.1.1 SCOAP3

The 'Sponsoring Consortium for Open Access Publishing in Particle Physics' (SCOAP3) converts literature in the field of High-Energy Physics to Open Access. It started its operations in January 2014 and aims to cover 4000 articles per year [8].

Initiated by CERN, SCOAP3 is now counting thousands of libraries as partners from more than 40 countries.

SCOAP3 is centrally negotiating with publishers and paying for the costs required to provide Open Access articles, while publishers reduce the subscription fees for the customers who contribute to SCOAP3.

Additionally to being Open Access, the article's copyrights stay with the authors and the applied CC-BY license allows text- and data mining.

The articles that are converted to Open Access are also accessible in the SCOAP3 Repository.

(See [7])

1.1.2 INSPIRE

Inspire (or INSPIRE-HEP), as shown in [9], is an Open Access digital library and the successor of the 'Stanford Physics Information Retrieval System' (SPIRES). It is run by the collaboration of CERN, DESY[3], Fermilab[4], IHEP[5], and SLAC[6] and interacts with the preprint repository arXiv.org, NASA-ADS[7], PDG[8] and HEPDATA[9].

INSPIRE is based on the open source digital library software Invenio [9]. In addition to the scientific content, INSPIRE provides information and features like citation metrics[10], author profiles, extracted plots and crowdsourcing for certain metadata fields[9].

[3] short for 'Deutsches Elektronen-Synchrotron' in Germany
[4] short for 'Fermi National Accelerator Laboratory' in the USA
[5] short for 'Institute of High Energy Physics' in China
[6] short for 'Stanford Linear Accelerator Center' in the USA
[7] short for 'NASA Astrophysics Data System'
[8] short for 'Particle Data Group'
[9] a project compiling a database of HEP experiments

1.1.3 CERN Document Server

The CERN Document Server (CDS), is CERN's institutional repository collecting papers, articles, and other form of media like images and videos published at CERN. Additionally, it is used to manage the CERN library process of rental and acquisition of material.

1.1.4 arXiv

arXiv, started in August 1991, is a highly-automated electronic archive for preprints of research articles in the fields of physics, mathematics, computer science, nonlinear sciences, quantitative biology and statistics. It is operated by the Cornell University Library with support form the arXiv Scientific Advisory Board, the arXiv Sustainability Advisory Group and numerous moderators.

Users of the arXiv service can access papers via the arXiv.org's web interface, while registered users can submit and update their papers to the repository. In case of an update, existing old versions remain available.

arXiv is an openly accessible, moderated repository which reviews the submitted articles in order to maintain content, which is relevant and of interest to the topics supported by arXiv.

By the time of 2014, the submission rate lies at more than 8000 articles per month [12].

(See [11])

1.2 Invenio

Invenio is a digital library and document repository software, which was first released in 2002 and started in the High-Energy Physics domain as an institutional repository and library system in the form of CERN's CDS and as a disciplinary repository in the form of INSPIRE.

The software covers every aspect of digital library management, starting at document ingestion, to classification, indexing and curation.

Invenio is currently co-developed by an international collaboration composed of CERN, DESY, EPFL[10], Fermilab and SLAC, to name some, and is used by about thirty institutions worldwide.

[10] short for 'École Polytechnique Fédérale de Lausanne'

In the production version of Invenio, MARCXML, which will be described in 2.1, is used as the native internal format.

(See [13])

2 Bibliograpic Metadata

Metadata is ”data about data”[14] and provides three different kinds of information [14]:

- Descriptive metadata: About specific data itself.
- Structural metadata: About the specification of the data structures.
- Administrative metadata: About technical information about the data structure.

Starting from there, bibliographic meta data serves to identify records and their versions, as well as to store any other source of information that makes working with the record easier [14].

2.1 MARCXML

MARCXML is part of the MARC ('MAchine-Readable Cataloging') based standards (see [16] for a list of other formats based on MARC), which was developed by Henriette Avram and a small team between 1965 and 1968 at the US Library of Congress [15]. Those MARC standard digital formats describe items that are supposed to be cataloged by libraries, such as books, and make them usable by computers.

MARC became the international standard for the dissemination of bibliographic data by 1973 [15].

Several version of MARC came up, with MARC 21 being the latest, created in 1999 [17].

MARCXML is an XML schema containing MARC data.

Since Invenio proposes a different, JSON based, internal format for it's future releases, MARCXML is not very important for this thesis. Nonetheless, the following aspects are worth noting.

2.1.1 Tags

In MARC, a tag is a three digit number and they represent certain fields in a record [18]. In Invenio for example, the tag 245 represents the title of the corresponding record.

Some tags can appear multiple times in a record.

2.1.2 Indicators

Each field can have two indicators, which are represented by a single digit. It is possible for a field to have zero, one or two indicators and it is also possible to just have the first or just the second one. A blank indicator is represented by the '#' character [18].

The meaning of indicators is field specific.

2.1.3 Subfield codes

The data in each field is divided into subfields, each of which is identified by a subfield code. Those codes are, in the MARC 21 standard, preceded by the '$' character. In general, every fields needs to have at least one subfield [18].

2.1.4 Example

Using the details view of www.inspirehep.net 's records[11], it is possible to look at the records metadata in various formats, which will give a better idea of how MARC and MARCXML looks like.

For the sake of this thesis, here an example for MARC and MARCXML in the way it is displayed by inspirehep, without showing actual data.

MARC

Listing 1: MARC format example

```
<0 digit inspirehep record_id> 001__ {{record_id}}
{{ record id }} 035__ $$9arXiv$$a{{oai:arXiv.org:id}}
...
```

The first nine digits of each line of MARC reference the internally used inspirehep ID, the following three digits determine the tag, succeded by the two possible indicators. In the case of inspirehep, a blank indicator is represented by the '_' character. The subfield codes and actual values are then shown as a single string, each subfield starting with two '&', one additonal character and the actual value of the field. See listing 1 as an example.

[11] Searching for a record, following the 'Detailed record' link, exposes multiple export options.

MARCXML

Listing 2: MARCXML format example

```
<collection xmlns="http://www.loc.gov/MARC21/slim">
    <record>
        <controlfield tag="001"> {{record_id}} </controlfield>
        <datafield tag="035" ind1=" " ind2=" ">
            <subfield code="9">arXiv</subfield>
            <subfield code="a"> {{oai:arXiv.org:id}} </subfield>
        </datafield>
    ...
    </record>
</collection>
```

The XML format is a bit easier for humans to read due to the provided semantic information. There is also a differentiation between controlfields and datafields, where the former doesn't include any indicators and subfields. This is not given in regular MARC. See listing 2 for a brief example of the general outline.

2.2 RecJSON

RecJSON is the internally used bibliographic meta data format of invenio v2. Taking the step away from a MARC based format towards an JSON format is rooted in these causes:

Using an internal JSON based format adds a layer of abstraction to the system. This would also be possible with a MARC based format, but JSON offers a wider range of tools, since the format is currently trending. This allows, most importantly, an easier handling of records.

For example, utilizing JSON grants access to using the following tools for invenio:

- elasticsearch[12]
- MongoDB[13]
- ...

(See the presentation [19].)

[12]"a search server based on Lucene"[21]
[13]"a document database using JSON"[22]

2.2.1 Structure

RecJSON follows the default JSON formula to construct objects and defines JSON objects for specific MARC fields.

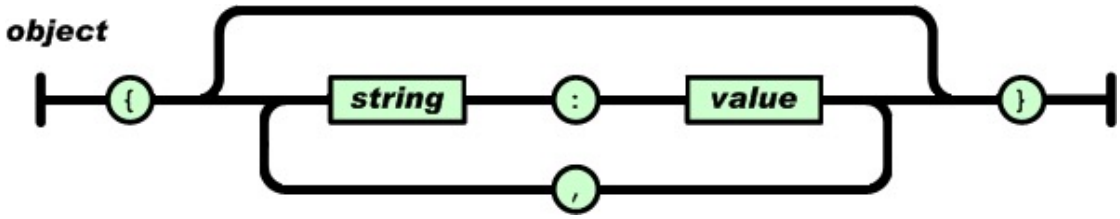

Figure 1: JSON object schema: Each JSON object is a comma separated list of a key, a colon, and a value, enclosed by curly brackets [20].

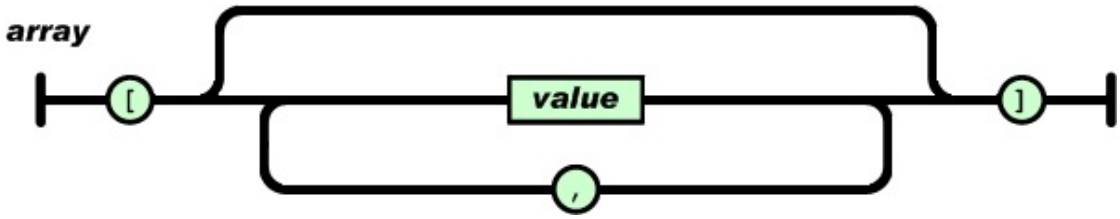

Figure 2: JSON array schema: A JSON array is a comma separated list of values, enclosed by square brackets [20].

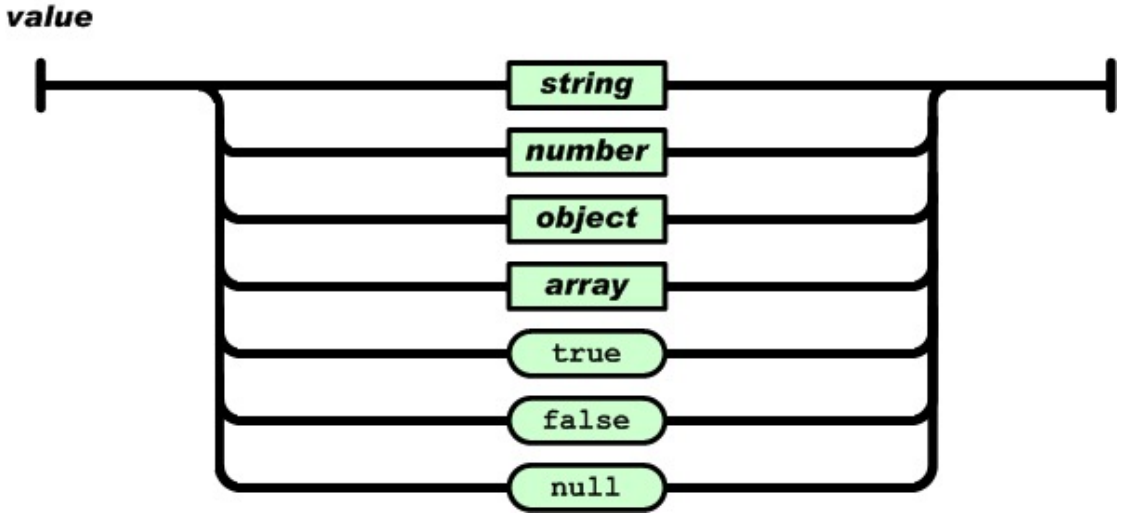

Figure 3: JSON value schema: A JSON value might be a string, a number, a JSON object, a JSON array, a Boolean or None [20].

Figure 1 shows the schema for every JSON object, which are a collection of key-value pairs. A key is expressed by a string, whereas a value can be everything depicted in figure 3. JSON objects, as seen in 2, are comma separated lists of values.

Considering the authors information in the meta data, the following transition is made: In MARC, authors are listed in the field 100 and as multiple occurrences of the field 700, with the subfield 'a' containing the name as a string and the subfield 'u' containing the affiliation of the author, as shown in listing 3.

Listing 3: MARC authors example, showing the first (field 100) and two additional authors (field 700)

```
<0 digit inspirehep record_id> 100__ $$aMustermann, Max$$uCERN
<0 digit inspirehep record_id> 700__ $$aDoe, John$$uAtlantis U.
<0 digit inspirehep record_id> 700__ $$aDoe, Jane$$uAtlantis U.
```

In RecJSON, the authors are shown in the "authors" key as a list of JSON objects applying the format shown in listing 4:

Listing 4: RecJSON authors example, showing the authors

```
"authors": [{"first_name": "Max",
             "last_name": "Mustermann",
             "full_name": "Mustermann, Max",
             "affiliation": "CERN"},
            {"first_name": "John",
             "last_name": "Doe",
             "full_name": "Doe, John",
             "affiliation": "Atlantis U."},
            {"first_name": "Jane",
             "last_name": "Doe",
             "full_name": "Doe, Jane",
             "affiliation": "Atlantis U."}
           ]
```

2.2.2 Example

Since JSON data structures are generally easy to read, the following example will be more elaborate to give a sense to the scope and content of Invenio's bibliographic meta data.

The example in 5 will provide an idea of a meta data file, without giving away any real content and without being complete regarding the possible keys. The values of each key are briefly explained between double curly brackets or use some real values, in case they delivered a better understanding.

Listing 5: RecJSON bibliographic metadata example

```
[{"comment": [{{ comment }}, ... ],
  "reference": [{"authors": {{ list of authors }} ,
                 "misc": {{ publication information }} ,
                 "year": {{ year of publication }} ,
                 ...},
                 ...
                ]
  "abstract": {"number": "arXiv",
               "summary": {{ summary from the records }} },
  "creation_date": {{ date }} ,
  "primary_report_number": {{ arXiv id or similar }} ,
  "subject": [{"source": "INSPIRE", "term": "General Physics"},
              ...],
  "physical_description": {"pagination": {{ pages number }} },
```

```
"number_of_citations": {{ citations }} ,
"title": {"title": {{ title }} },
"persistent_identifiers_keys": ["recid",
                                "system_control_number"],
"files": [{"comment": null, "status": "",
           "magic": ["PDF document, version 1.4",
                     "application/pdf; charset=binary",
                     "PDF document, version 1.4",
                     "application/pdf; charset=binary",
                     "application/pdf"],
           "description": null,
           "url": {{ url to record }} ,
           "eformat": {{ format (pdf, ...) }} ,
           "version": {{ version number }} ,
           "full_name": {{ name of the file }} ,
           "size": {{ file size }} ,
           "full_path": {{ path to file on disk }} ,
           "name": {{ name of record }} },
           ...
         ],
 "system_control_number": [{"institute": "INSPIRETeX",
                            "value": {{ control number}}},
                           {"institute": "arXiv",
                            "value": {{ control number }} },
                            ...
                          ],
 "filenames": [{{ file name }} , ...],
 "source_of_acquisition": {"source_of_acquisition": "arxiv"},
 "number_of_reviews": {{ number of reviews }}
 "version_id": {{ version id }},
 "authors": [{"first_name": {{ first name }},
              "last_name": {{ last name }},
              "full_name": {{ combined, comma separated }} },
             ...
            ],
 "number_of_authors": {{ number of authors }},
 "license": {"imposing": "arXiv", "url": {{ arXiv url }} },
 "number_of_comments": {{ number of comments }},
 "title_additional": {"title": {{ additions }} },
 "recid": {{ recid }},
 "collection": [{"primary": "CORE"},
                {"primary": "arXiv"},
                {"primary": "Citeable"},
```

```
                    {"primary": "HEP"}],
    "filetypes": ["pdf"],
    "prepublication": {"date": {{ date}} }
  }
]
```

3 Merging

Merging, in the sense of the word, means the combination of multiple objects to a single entity [25]. In the environment of 'version control systems'[14] (also called 'revision control systems'), a merge, as shown in figure 4, mostly refers to the combination of files with the same origin to a new, single file [26].

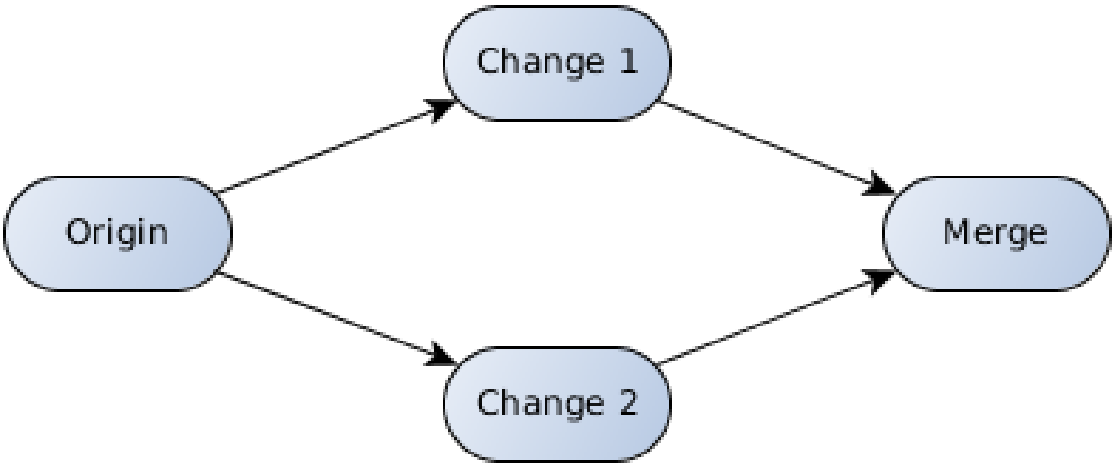

Figure 4: A basic merge: Two changed files of the same origin are combined to a new file.

This section will discuss general principles of merging records and will then show how it differs from the merging of the repositories metadata structures.

3.1 General Merging

The scenarios encountered by the document repository systems are similar to those of revision control systems, for example git[15], in that a record can be affected by the changes made by catalogers and the updates coming from the publishers. Due to this analogy, this paper will approach the basics of merging in the way a revision control system handles them. From the point of view of revision control systems a merge is an operation that combines two files with a common origin to a new file, incorporating changes of

[14] 'a system that records changes to a file or set of files over time'[23]
[15] a revision control system originally developed by Linus Torvalds[24]

both. This happens mostly, when two or more contributors make changes to the same file [26].

3.1.1 Three-Way Merge

For this thesis, the 'Three-Way Merge' will be the basis of further considerations. A 'Three-Way Merge' considers the differences between the two files A and B, as well as their 'latest common ancestor' (LCA) C.

The 'latest common ancestor', sometimes called 'nearest common ancestor' is the entity in the history from which the other files emerged. Considering figure 4, the entities label 'Change 1' and 'Change 2' emerged from the entity label 'Origin', therefore, 'Origin' is the 'latest common ancestor' for 'Change 1' and 'Change 2'.

There are three possible states for a section in the files A, B and C:

1. The section is the same in all three files.
2. The section is the same in two of the three files.
3. the section differs in all three files.

The first case represents the situation when the sections in a file were not changed. The second case can mean one of the following things:

- The section in C and A or B have the same content. This means that file A or B was changed and this change can be taken over to the resulting file.
- The section in A and B have the same content. In this case A and B were changed in the same way, so the content can be taken for the resulting file.

The last case means that files have been edited in the same section, but don't agree on the content, this is considered to be a conflict.

State	C	A	B	Result
1	'foo'	'foo'	'foo'	Take C, A or B, no change, no conflict
2	'foo'	'bar'	'bar'	Take A or B, agreeing change, no conflict
2	'foo'	'foo'	'bar'	Take B, update by B, no conflict
2	'foo'	'bar'	'foo'	Take A, update by A, no conflict
3	'foo'	'bar'	'baz'	Conflict

Table 1: The three possible states for each section in the files C, A, and B.

Table 1 shows a summary of all possible sates for each section in case of a Three-Way Merge.

The Three-Way Merge will apply all conflict-free changes to the latest common ancestor and will survey a higher instance for a resolution of the conflicts. In most cases, this higher instance is a human being.

(see [26].)

3.1.2 Finding Differences

The most important part of useful Three-Way Merge is the way it finds the differences between the files C and A, and C and B. Poorly recognized differences can lead to:

- Avoidable conflicts: Recognizing an insertion as a change in a text file might entail further changes, which didn't really happen, thus introducing more conflicts.
- Missing semantic information: Multiple changes, which are not recognized as a semantic block might lead to incomplete conflicts, which can be problematic in automatic resolution processes.

Finding differences in files strongly depends on the present data structure. Since this thesis is based in the Python programming environment, we are going to consider the following data structures: Lists, Sets and Dictionaries. Python's dictionaries can depict XML, JSON, or other key value data structures, whereas lists can represent unstructured data like text files. Sets are special in the sense that they are generally unstructured, but their items are unique.

Data types which are not a form of a collection (for example integers or strings), are compared directly and not investigated any further.

Structured Data Structured data like XML, JSON or other forms of key value data are, in principle, easier to handle than unstructured data. Finding differences in a JSON or XML file for example comes down to traversing its keys and comparing the values, which can be done by a simple recursive algorithm. See 4.1.2 for a recursive algorithm to extract differences from structured data. The way of extracting differences from lists depends on the way they are interpreted. The index of each item in a list can have a semantic meaning, which turns a list into structured data, therefore, again, leading to comparing the values for each index.

Unstructured Data In case that the index of an item in a list has no semantic meaning, extracting differences becomes more complicated. In this case the value of the items in

the list and their order determine the semantics, which leads to the fact that a simple item by item comparison is no longer suitable. One approach of finding the differences in two lists is solving the 'longest common subsequence problem', which tries to find the longest subsequent list of items that is present in both files. This problem will be discussed to a greater extend in the next paragraph.

Longest common subsequence problem

The 'longest common subsequence (LCS) problem' describes the problem of finding the longest subsequence common to two given sequences. It is a classical computer science problem and has applications in data comparison, bioinformatics and other fields [27].

A simple algorithm to solve the LCS problem can be described by the following recursive algorithm, taking a few considerations into account (see [28]):

1. In case that both sequences A and B start with the same letter, those letters can be removed, since they are certainly part of the LCS. Thereby starting the LCS algorithm again with a reduced sequence.
2. In case that the letters are not equal, it needs to be decided which sequence should be shortened by the first letter. Since we are searching for the longest common subsequence, the result of trying both possibilities can be evaluated against the resulting length.

Since computing the length of a LCS is simpler than the LCS itself in a recursive way, this will be used as a starting point for further advancements which will then be able to return the LCS.

Listing 6: Longest common subsuquence algorithm for calculating the length of the LCS using a recursive approach.

```
def lcs(A, B):
    if A == '' or B == '':
        return 0
    elif A[0] == B[0]:
        return 1 + lcs(A[1:], B[1:])
    else:
        return max(lcs(A[1:], B), lcs(A, B[1:]))
```

The approach displayed in listing 6 performs badly. Assuming two sequences without a common subsequence, the last line will always be executed, which leads to a complexity of O(2^n).

Luckily, algorithms of the recursive form present for the given LCS function are prone

for 'dynamic programming'[16], which will reduce the complexity for the sake of space.

In this case, applying dynamic programming principles result in the algorithm shown in listing 7:

Listing 7: Longest common subsequence algorithm for calculating the length of the LCS using dynamic programming

```
def lcs(A, B):
    M = create_storage(A, B)
    A = ' ' + B
    B = ' ' + B
    for i, a in enumerate(A[1:], 1):
        for j, b in enumerate(B[1:], 1):
            if a == b:
                M[i][j] = 1 + M[i-1][j-1]
            else:
                M[i][j] = max(M[i-1][j], M[i][j-1])
```

Traversing the resulting matrix of this algorithms in the way described in listing 8:

Listing 8: Longest common subsequence algorithm for traversing the LCS matrix for finding the actual LCS.

```
def traverse(A, B, M):
    sub_sequence = ''

    while i < len(A) and j < len(B):
        if A[i] == B[j]:
            sub_sequence += A[i]
        elif M[i-1][j] <= M[i][j-1]:
            i += 1
        else:
            j += 1
```

will return the longest common subsequences with a complexity of O(n*m).

There are of course more advanced solutions to this problem, one of them is going to be introduced in a later section.

See chapter 4.3.2 for more information on how to compute the actual changes from this longest common subsequence.

[16]programming approach to solve and combine overlapping subproblems to solve the original problem[27]

3.1.3 Other Merging Algorithms

Due to certain specific problems, like indistinguishable LCAs, different kinds of merge algorithms evolved. This section does not want to give an in depth inside in those algorithm, it just aims to provide a slightly broader view of the topic of merging, showing one known problematic case and a different approach.

Recursive Three Way Merge One important factor for a Three-Way Merge to be successful is a valid latest common ancestor. This LCA might be, for example in the case of the 'criss-cross merge' [29], indeterminable. Since [29] shows, that there are at most two candidates for the LCA, the recursive Three-Way Merge tries to merge those two candidates first. This process might repeat itself, hence the name recursive Three-Way Merge.

Weave Merge The Weave Merge abdicates the need of a LCA and tries to produce the merged file by considering the following information line by line:

- Was this line deleted in a derivate version of the file?
- Which line precedes it?
- Which line follows it?

Based on this information, the Weave Merge makes the following decisions: If the line was deleted in either derivate of the file, it will not be included in the final merge.

The lines are then sorted in a way that every line follows every line that preceded it, and precedes every line that followed it, at some point in history.

Each line that doesn't end in a total ordering following the mentioned constraints is considered a conflict.

(See [26])

3.2 Merging of Bibliographic Metadata

Merging of bibliographic metadata differs from the merging of text files, due to the fact that they are structured. They are not a line based collection of information, they are a collection of key value pairs. Due to this fact simply solving the LCS problem might, depending on the representation, lead to bad results. The structured nature of bibliographic metadata is an aid that should be used.

3.2.1 Representation of metadata

Bibliographic metadata in the Invenio environment or the library environment is represented as MARCXML or JSON files. Since those JSON files are going to be the future representation format of the metadata, this thesis will consider the metadata to be JSON files.

So generally speaking the bibliographic metadata is a tree structure, or in Python terms, a nested combination of dictionaries, lists and sets. Those structures, except for the list, make it easy to determine differences.

Following the tree structure down to its leaves (the sequence of keys necessary for the traversal will be called 'path' throughout the thesis), there are four options:

1. The value for the path is the same.
2. The value for the path differs.
3. The value for the path doesn't exist in the old record.
4. The value for the path doesn't exist in the new record.

Listing 9: Example of the four different states for key value pairs of metadata.

```
Data structure 1 (DS1)
{'foo': {'the_same': 'Same value',
         'different': 'Value 1'},
 'bar': {'deleted': 'Deleted later on'}
}

Data structure 2 (DS2)
{'foo': {'the_same': 'Same value',
         'different': 'Value 2'},
 'bar': {'added': 'Added'}
}
```

Regarding listing 9 above, the following table (Table 2) shows the correlation between the key sequence and the four possible options.

Option	key sequence (path)	Reason
1	'foo', 'the_same'	'Same value' == 'Same value'
2	'foo', 'different'	'Value 1' != 'Value 2'
3	'bar', 'added'	The key sequence and value doesn't exist in DS1
4	'bar', 'deleted'	The key sequence and value doesn't exit in DS2

Table 2: Mapping of the example from listing 9 to the four possible state for key value pairs in the metadata.

For those cases there is no need for a diff-like algorithm that tries to determine a longest subsequence in order to get some coherent changes. Changed content in the same path down the data structure is sufficient in this case.

List structures are the exception here. Considering a list as a structure where the order plays an important role (Table 3), it becomes apparent that an index-vise comparison is not suitable enough. In this case, a sequence based difference algorithm like the one used in unix diff would be appropriate.

First	Second
Hello,	Hello,
This is some text in line 2.	This is different text in two lines.
Bye.	Bye.

Table 3: Example of two files, where the ordering of the lines has semantic meaning.

But there are at least two more interpretations for a list: It could just be a collection of elements, where the order of the elements has no semantic meaning, as shown in table 4. In this case a index by index comparison is sufficient. Furthermore, each element in the list could have a specific meaning, just like a key in the dictionary (see table 5). This case would also not need to be checked by a sequence based difference algorithm.

First	Second
Random data 1	Random data 2
Random data 3	Random data 3
Random data 4	Random data 5

Table 4: Example of two files, where the ordering of the lines has no semantic meaning.

First	Second
John Doe	John Do
High Street	High Street
unknown@mail.com	known@mail.com

Table 5: Example of two files, where the index has semantic meaning.

3.2.2 Conflicts

Once that the patches are extracted, determining a conflict is very similar to a regular merge tool for some sort of text file: In case that both files make a change in the same section (say path in the tree structure) and in the case that this change is not the same, there is a conflict.

Another difference is the fact that also changes to a 'super path' conflict to the changes of the child elements. For example: The patch that deletes one object will conflict with those patches, that change those objects elements and sub elements. See listing 10 for an example. In the displayed case, the path 'author', where the deletion happens, is the super path of 'author.last_name'.

Listing 10: Conflicting changes, one regarding the super path of the other.

```
LCA:
{'author': {'first_name': 'John', 'last_name': 'Do'}}

Changed Data Structure 1:
{}

Changed Data Structure 2:
{'author': {'first_name': 'John', 'last_name': 'Doe'}}

Patches LCA -> CDS1:
('remove', '', [('author',
                 {'first_name': 'John', 'last_name': 'Do'})])

Patches LCA -> CDS2:
('change', 'author.last_name', ['Do', 'Doe'])
```

4 Automated Merging Tool

This section is going to be about the design and realization of the automated merging tool, which should be capable of extracting, altering and merging patches.

For the sake of clarity, a little foreshadowing: The dictdiffer module will be amended and host the automated merging module, since it provides the functionality of extracting differences in data structures.

Therefore, this section will be divided into the following parts:

- Current State Analysis: Analyzing the currently used version of the dictdiffer module to get a starting point.
- Conception of the new dictdiffer: Starting from a desired goal or a known problem, possible solutions will be introduced and discussed.
- Realization of the proposed changes to the dictdiffer module.
- Conception of the automated merging tool: After knowing about the current state of the dictdiffer and the changes that will be made to it, the concept of the automated merging tool will be discussed.
- Realization of the proposed automated merging tool.
- Theoretical evaluation.

4.1 Current State Analysis of invenio.dictdiffer

According to the documentation, Invenio's dictdiffer is a module that "helps you diff and patch dictionaries"[30]. The module is currently under development and is used not only in the scope of Invenio.

The dictdiffer module follows a functional programming approach, meaning that the functions of the module avoid making changes to the present data structures[31]. Every needed information is passed as an argument to the specific function.

This design allows a simple interface for the user, containing only four functions to call.

The analysis executed in this section was done by reading the source code of dictdiffer version 0.3.0.

4.1.1 Patch Format

Invenio's dictdiffer currently extracts the three different kinds of patches displayed in listing 11:

Listing 11: The outline of the three dictdiffer patches

```
('add', <path>, [(<index/key>, <value of addition>),...])
('remove', <path>, [(<index/key>, <value to remove>),...])
('change', <path>, (<old value>, <new value>))
```

Generally speaking, a patch is a triple (3-tuple) consisting of the following parts.

1. The type of patch ('change', 'add', 'remove'), which is important later to decide how the patch gets handled.
2. The path, which determines where the changes are supposed to be applied.
3. The actual changes.

There are two notable differences between the add/remove and the change patches:

1. In case of the add/remove patches, the path to the actual change is split between the second and the third part of the patch, meaning that the 'index/key' value shown in listing 11 is part of the path.
2. Add/remove patches contain a list of additions/deletions, whereas a change patch only handles one change.

In the following part of this thesis, the different kinds of patches will be distinguished by their action, meaning that a patch, which describes an addition, will be referred to as an addition patch. The same logic applies for the other two patches.

4.1.2 diff

Listing 12: diff function definition

```
def diff(first, second, node=None, ignore=None)
```

The basic diffing function (Listing 12). It takes two manageable data structures and returns a generator object of a list of tuples which describe the changes between the data structures. Those generators are special class which return an iterator, that returns a stream of values and doesn't store them in memory [31] (See 4.3.1 for more information about generators). Additionally, the diff function provides a parameter to ignore certain

keys and a node parameter, which is used in the recursive process to keep track of the current position in the data structure.

The diff function does the following:

- Distinguishing the current node in the data structure and converting it into a special string notation, which is used as the path of the patch if it is applicable. Applicable means, that the key sequence consists of strings only.
- Extracting the differences between the nodes depending on the three supported data structures: lists, dictionaries and sets.
  - In order to get the differences, the compared data structures need to have the same type at the current node.
  - The differences are then grouped in the following categories: intersections, additions and deletions.
  - It is possible that there are no differences. Either because the sub-structures are the same, or because they differ in their types and can not be compared.
- Differences that are in the categories additions and deletions are directly presented as a patch of the according format. (More on this in the next section). Differences of the kind of an intersection are further examined by passing them to the diff function with the previously extracted node.
- In case that no differences could be found because the sub-structures differed, a change patch will be returned.

See figure 5 for a more visual explanation.

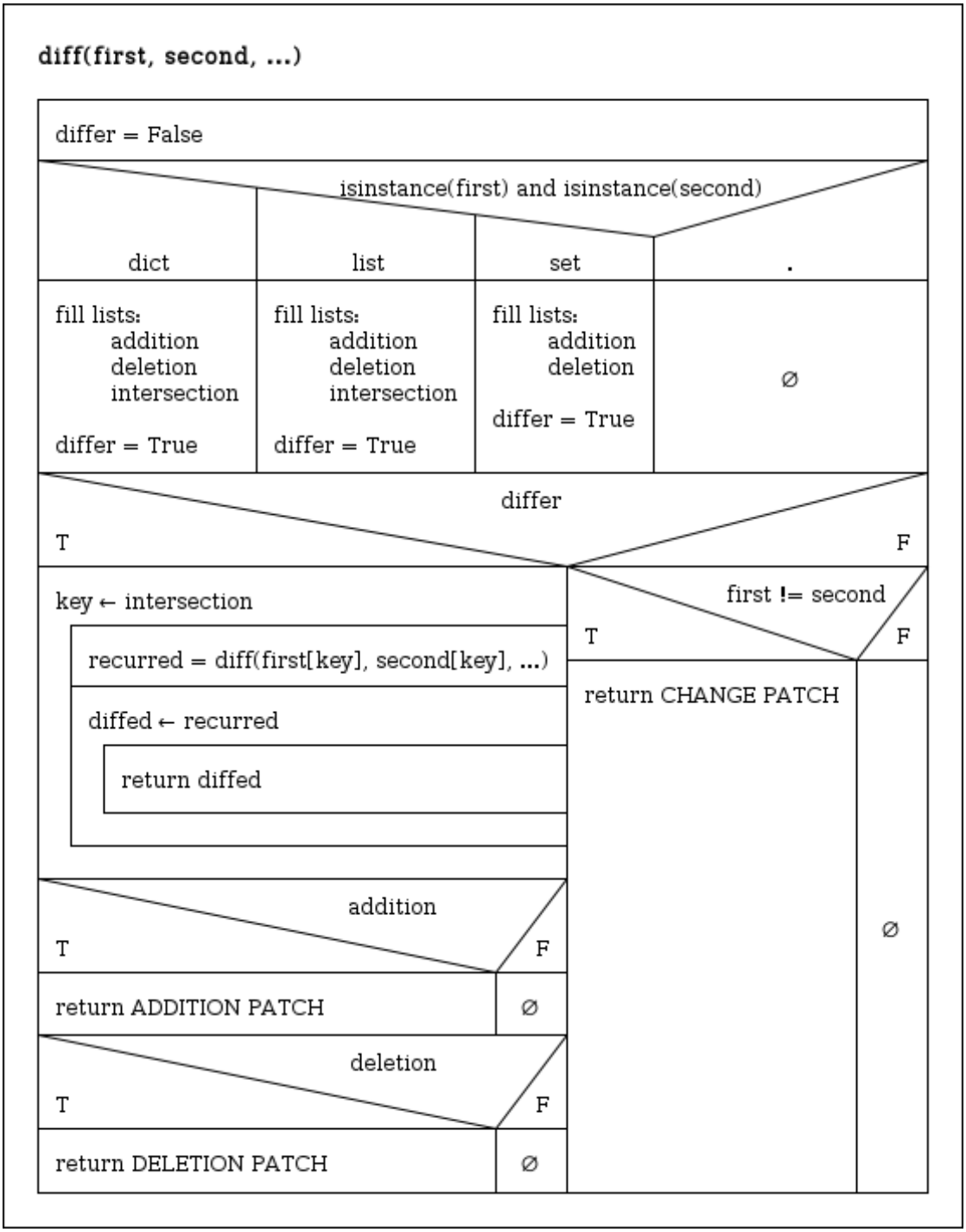

Figure 5: Structure chart of the original diff function.

4.1.3 patch

Listing 13: patch function definition

```
def patch(diff_result, destination)
```

This function (listing 13) applies the extracted patches to the given data structure 'destination', which is not changed in place: A copy is created and then altered.

To do so, the patch function iterates the result of the diff function and calls a sub-function according to the type of the present patch.

Those three sub-functions differ in details, but share the following basic outline:

- Fetch the specified position in the data structure by following the path of the patch.
- Determine the type of the sub-structure at the specific position.
- Make the changes according to the determined type.

4.1.4 swap and revert

Listing 14: swap function definition

```
def swap(diff_result)
```

The swap function (listing 14) swaps the diff result, meaning it changes additions to deletions and vice versa, while changing the values accordingly. Change patches remain change patches, but switch the values. See listing 15.

Listing 15: swap function example

```
swap( ('add', 'foo', [('bar', 'baz')]) )
-> ('delete', 'foo', [('bar', 'baz')])

swap( ('delete', 'foo', [('bar', 'baz')]) )
-> ('add', 'foo', [('bar', 'baz')])

swap( ('change', 'foo.bar', ['baz', 'bay']) )
-> ('change', 'foo.bar', ['bay', 'baz'])
```

Listing 16: revert function definition

```
def revert(diff_result, destination)
```

Revert (listing 16) is basically a patch with a preceding swap. It will revert the changes applied to a data structure by applying the swapped version of the existing patches.

The swap and revert functions will not be further discussed in this thesis, since they are not particularly interesting for the merging process.

4.1.5 Noteworthy

There are two observations of the current state of the dictdiffer module that are worth mentioning:

The first is the fact that it lacks the possibility to maintain any information about the order of the applied changes since it just stores the changes in three different lists and returns their content one after the other. This approach is generally fine for dictionaries and sets, but only works in some cases for lists, where insertions, changes and deletions could alternate at random.

The second observation is that the provided patches are returned in a way that they 'know' what happened before they are applied. The path in a patch contains the index required to apply the patch at the right position if every other patch was applied before it. While this is fine for applying all extracted patches, it holds some limitations for the scenarios of removing unwanted patches and finding differences.

Furthermore it should be emphasized that the extraction process is not adjustable. Additions and deletions are presented as a whole, meaning the object with all its contained sub-objects, not introducing possible enclosing data structures in steps. Changes on the other hand are always as 'detailed' as possible. In this scenario, 'detailed' means that the difference extraction process traverses the data structure to the point where the amendment occurs, never stopping at a more general object before.

4.2 Conception of invenio.dictdiffer

Considering the current state of the dictdiffer module and the automated merging that is the goal of this thesis, a few changes need to be made. Those changes will mostly deal with the limited approach for extracting differences in lists and the fact that the extraction process is currently not adjustable.

4.2.1 Using dictdiffer

Since this thesis and the need for the functionality it is supposed to satisfy originated in the Invenio environment, it appears to be reasonable to use and improve as much existing functionality as possible. For that reason the work in this thesis will build upon the patch extraction process provided by the dictdiffer module. Since this module is already used, not only by Invenio, maintaining backward compatibility in the sense of results and the established interface are a strong requirement.

For the sake of coherence, the proposed changes should also fit in with the existing style and programming paradigms, namely the functional approach of the current implementation.

4.2.2 General requirements

Generality To allow a wider array of possible use cases, addressing the problem in a reasonable general way seems to lead to a better final product.
Bibliographic meta-data is therefor interpreted as nested collections. More precisely a combination of lists, dictionaries and strings. The generalization of this is a combination of any Python built-ins and collections and even custom objects. This approach might lead to features, which are not needed for the integration of the automated merging tool into the Invenio environment, but will hopefully lead to a better final project with a wider variety of possible use cases.

Customizability The scope of this thesis is to provide a automatic merging tool for the metadata updates from the repositories SCOAP3, Inspire, CDS and arXiv, which should be easily customizable in case of future additions. Or, regarding the generality requirement: Providing an easy way to use it in different environments. Additionally, there should be a default configuration that just works out of the box.

4.2.3 Depth of Extraction

In the current state, the diff functions difference extraction is not configurable. An addition will always end up in an addition patch containing the added elements and a change patch will always just handle the relevant element. Listing 17 shows a result of applying the diff function to two data structures. It becomes apparent, that the value assigned to the 'author' key is not handled by a single change patch, but by an addition and a change patch.

Listing 17: Example of applying the dictdiffer.diff function to two data structures.

```
Original Data Structure
{'author': {'name': 'Jo'}}

Changed Data Structure
{'author': {'name': 'Bob', 'street': 'High'}}

Actual patches:
('change', 'author.name', ('Jo', 'Bob'))
('add', 'author', [('street', 'High')])

Different possible behavior:
('change', 'author', ({'name': 'Jo'},
                      {'name': 'Bob', 'street': 'High'}))
```

This behavior is reasonable, but leads to the following problem for a rule based merging tool: Since the latest common ancestor, from which the patches are extracted, can have various different forms, ranging from a very similar data structure to an empty dictionary, the extracted patches would not provide the needed consistency, which would lead to the necessity of many different automatic resolution functions, which basically all handle the same conflict.

Listing 18 illustrates the problem of different latest common ancestors. In case of the same two changed data structures, a different LCA leads to strong aberrant patches: In the first example the differences from the LCA to the changed data structures are handled with two patches, one addition and one change patch. In the second example, utilizing and empty LCA, the changes are handled by one addition patch.

Listing 18: Illustraion of the problem occurring due to different latest common ancestors.

```
Latest Common Ancestor
{'author': {'name': 'Jo'}}

Changed Data Structure 1
{'author': {'name': 'Bob', 'street': 'High'}}

Changed Data Structure 2
{'author': {'name': 'John', 'street': 'High'}}

Patches LCA -> CDS 1
('change', 'author.name', ('Jo', 'Bob'))
('add', 'author', [('street', 'High')])
```

```
Patches LCA -> CDS 2
('change', 'author.name', ('Jo', 'John'))
('add', 'author', [('street', 'High')])


Latest Common Ancestor
{}

Changed Data Structure 1
{'author': {'name': 'Bob', 'street': 'High'}}

Changed Data Structure 2
{'author': {'name': 'John', 'street': 'High'}}

Patches LCA -> CDS 1
('add', '', [('author', {'name': 'Bob', 'street': 'High'})])

Patches LCA -> CDS 2
('add', '', [('author', {'name': 'John', 'street': 'High'})])
```

In order to provide the needed consistency, this thesis proposes the following solution: Controlled by a flag (called 'expand'), the diff function tries to expand the patches to be as 'detailed' as possible in the patch extraction. In other words: an addition patch doesn't simply add the object as a whole, but adds the empty data structure first and then the contained elements, as shown in listing 19.

Listing 19: Example of the difference extraction using the 'expand' flag, which leads to more specific patches.

```
Latest Common Ancestor
{}

Changed Data Structure 1
{'author': {'name': 'Bob'}}

Patches LCA -> CDS 1 using 'expand' flag
('add', '', [('author', {})])
('add', 'author', [('name', 'Bob')])
```

From there we add another object that determines if the extraction process should stop at a given depth. This object will be called 'path limit'.

Listing 20: Example of the difference extraction utilizing the 'path limit' object, which will stop the extraction process at a given path in the data structure.

```
Path Limit: 'author'

Latest Common Ancestor
{'author': {'name': 'Jo'}}

Changed Data Structure 1
{'author': {'name': 'Bob'}}

Patches LCA -> CDS 1
('change', 'author', ({'name': 'Jo'}, {'name': 'Bob'}))
```

Since 'author' was the path limit for the example in listing 20, the patch extraction process could not traverse the data structure to the level of 'author.name', therefore providing a change patch that changes the dictionaries.

With those adjustments, the extractions process can be controlled to provide consistent patches for the following resolution.

4.2.4 Extensible Difference Algorithms

Currently, dictdiffer.diff supports three different data structures: dictionaries, sets and lists. The extraction of differences for the former two is rather straight forward: For dictionaries, the keys either don't exist in one of the data structures, which results in an addition or deletion patch, or they exist in both and the values differ or are the same, which results in a change patch or nothing. For sets, the values are unique, so there can only be additions or deletions,which can be recognized using set operations. In the case of lists, the extraction can get a bit more complicated.

The current difference algorithm for lists works as illustrated in listing 21: Given two lists OLD and NEW. Every element in NEW with an index greater than OLD is an addition, every index missing in NEW is a deletion and every changed element with an index in both lists is a change. This interpretation of a list ignores the consecutive nature of the elements of a list, which might be needed to be preserved.

Listing 21: Illustration of the old list differences extraction process.

```
OLD
[3]

NEW
[1, 2, 3, 4, 5]

Patches
('change', [0], (3, 1))
('add', '', [(1, 2), (2, 3), (3, 4), (4, 5)])
```

Considering that the sequence of elements in a list is important, the following patch extraction (listing 22) might feel more correct for a human:

Listing 22: Illustration of the proposed list differences extraction process.

```
OLD
[3]

NEW
[1, 2, 3, 4, 5]

Patches
('add', '', [(0, 1), (1, 2), (3, 4), (4, 5)])
```

It seems to make sense (probably in the sense of the ”gestalt principles”[36], which is utilized by the later described 'gestalt pattern matching' algorithm), that the elements 1 and 2 are added before, and the elements 4 and 5 are added after the existing element 3. Finding those kinds of differences can be solved by an algorithm which solves the 'longest common subsequence problem'. One notable implementation of those algorithms is Linux' diff3 command. Unfortunately, those algorithms can be unreliable (see [37]), which leads to the conclusion that providing the user the possibility to use their own, custom algorithm is more useful than deciding on one at this point of the thesis.

This means, that the dictdiffer.diff function needs a configuration parameter, which allows the user to pick the right extraction algorithm for the data structure at hand. Currently, the diff function calls Python's isinstance function on the first and second variables to determine which of the three given difference algorithms it should apply. The flaw in this approach is, that it's hard coded.

In order to solve this issue and make the process more versatile, this thesis proposes a dictionary parameter, which contains a mapping from a path or type to a difference algorithm as well as a default key (see listing 23).

Listing 23: Lookup dictionary object for the difference algorithms.

```
{<path>: diff_class1,
 <type>: diff_class2,
 'default': [diff_class1, diff_class2, ...]}
```

Getting a difference algorithm would then be a three level cascading process:

1. Decide on the algorithm based on the path in the data structure, which would be the most precise decision, since the path in the data structure is unique compared to the type or the instance.
2. Decide on the algorithm based on the type of the structure.
3. The defaults. The 'default' key provides a list of possible difference classes, which would provide the functions extract and is_applicable. This is_applicable function can use isinstance to determine if the algorithm can be used on the data structures.

To extend this dictionary parameter, the difference algorithm would need to be wrapped in the form shown in listing 24:

Listing 24: Class template to implement a custom difference extraction algorithm.

```
class <NAME>Extractor(object):
    @classmethod
    def is_applicable(cls, first, second):
        # do the required kind of type checking here

    @classmethod
    def extract(cls, first, second):
        # do the difference extraction here
```

Another detail that needs to be addressed in order to make additional difference algorithms work is the fact that the extracted differences do not preserve order. They just get filled in the three possible lists addition, deletion and intersection (see figure 5).

To provide an order, the difference algorithms won't return the three values of addition, deletion and intersection, but return a single list of 'meta patches'. Those patches are a quintuple (5-tuple) consisting of the fields shown in listing 25:

Listing 25: Schema of the 'meta patch' format used by the extraction algorithms.

```
(<patch type>, <old path>, <new path>, <old value>, <new value>)
```

The 'patch type' field works in analogy to the patch type of the real patches, 'old path' and 'new path' are the path of the data structure affected by the change. The same goes for 'old value' and 'new value'. The latter two are required for set like structures, which

do not provide direct access via indexes or keys. They also allows an easier recursive traversing of the data structure, since the 'old path' and 'new path' values will differ from the actual indexes, due to the fact that they need to be extracted in a way consistent to the recent dictdiffer patches.

Listing 26: Example of the patch extraction process, focussing on the indexes diverging from their 'actual' value.

```
a = [    1, 2,     4]
b = [0, 1, 2, 3, 5]

Extracted patches:
('add', '', [(0, 0)])
('add', '', [(3, 3)])
('change', 4, [4, 5])
```

Considering listing 26 above, the indexes in the patches are 0, 3 and 4, whereas, considering each change on it's own, they would be 0, 2 and 2. This is problematic in case of a change patch with the 'expand' flag set. The 'diff' method would need to further investigate the changes and therefore needs to traverse the data structure (see 4.3.1). Given only the index 4, it would need to know about the original index 2 in list 'a', it would therefore need to keep track of all the previous additions and deletions. Thus, it is easier to utilize the 'old val' and 'new val' objects.

Those 'meta patches' are proposed as the return values for the custom Extractor classes instead of letting each class provide the correct patches immediately, since the final form of the patches might change over time, and then there is only one point in the code to do so.

This change unfortunately changes the patch extraction in general. It makes it hard to maintain the old structure of the additions, deletions and intersections lists, which leads to addition and deletion patches without a collection of changes for one path (see listing 27 for the resulting new patch format).

Listing 27: Illustration of the difference between to old and new patches. Due to the proposed changes, addition and deletion patches will not contain a list of alterations anymore.

```
Old Style Patch
('add', '', [(0, 1), (1, 2), ...])

New Style Patch
('add', '', [(0, 1)])
('add', '', [(1, 2)])
...
```

To maintain the old style, the extractors would have to keep track of their extractions first, and then return the corresponding patches in chunks. But since the old distinction of patches in additions, deletions and intersections can not be maintained because of the necessity of an order anyways, and since the new style is not affecting the other functionality of the dictdiffer module, it will be kept this way.

It should be noted, that the changes should be kept inside a list to not break backward compatibility, but this should be adjusted with a future release after a deprecation warning for some time.

4.3 Realization of invenio.dictdiffer's changes

The following section illustrates some noteworthy implementation details of the amendments proposed in the previous section.

4.3.1 Changes to invenio.dictdiffer.diff

Invenio.dictdiffer's diff function will be the subject to most changes and it starts with amending the parameter list as displayed in listing 28:

Listing 28: Proposed diff function definition.

```
def diff(first, second, node=None, ignore=None,
         extractors=None, try_default=True,
         expand=False, path_limit=None):
```

The changes are straight forward in following the proposed changes in the previous section, with the notable exception of the 'try_default' flag, which takes part in managing the available extractors. The proposed 'default' key in the extractors dictionary would require more execution time than the rest of the options, since it would call Python's 'isinstance' method. To speed up this process, that lookup can be disabled.

The next amendment deals with the acquisition of the suitable extractor:

Listing 29: get_extractor function to choose an extraction algorithm according to the situation at the current position in the data structure.

```
def get_extractor(extractors, first, second, node, try_default):
    extractor = (extractors.get(tuple(node)) or
                 (extractors.get(type(first))
                  if type(first) == type(second)
                  else None))

    if extractor:
        return extractor
    else:
        if try_default:
            for extractor in extractors['default']:
                if extractor.is_applicable(first, second):
                    return extractor
```

This implementation in listing 29 follows the suggested structure of a cascading approach, by first querying an extractor for a given path, then for a type and at last by the 'is_applicable' method. The default return value of this function, even though not explicitly stated, is None if no extractor could be found.

From this point in the code, the program flow described in the figures 6, 7, 8 and 9 is utilized:

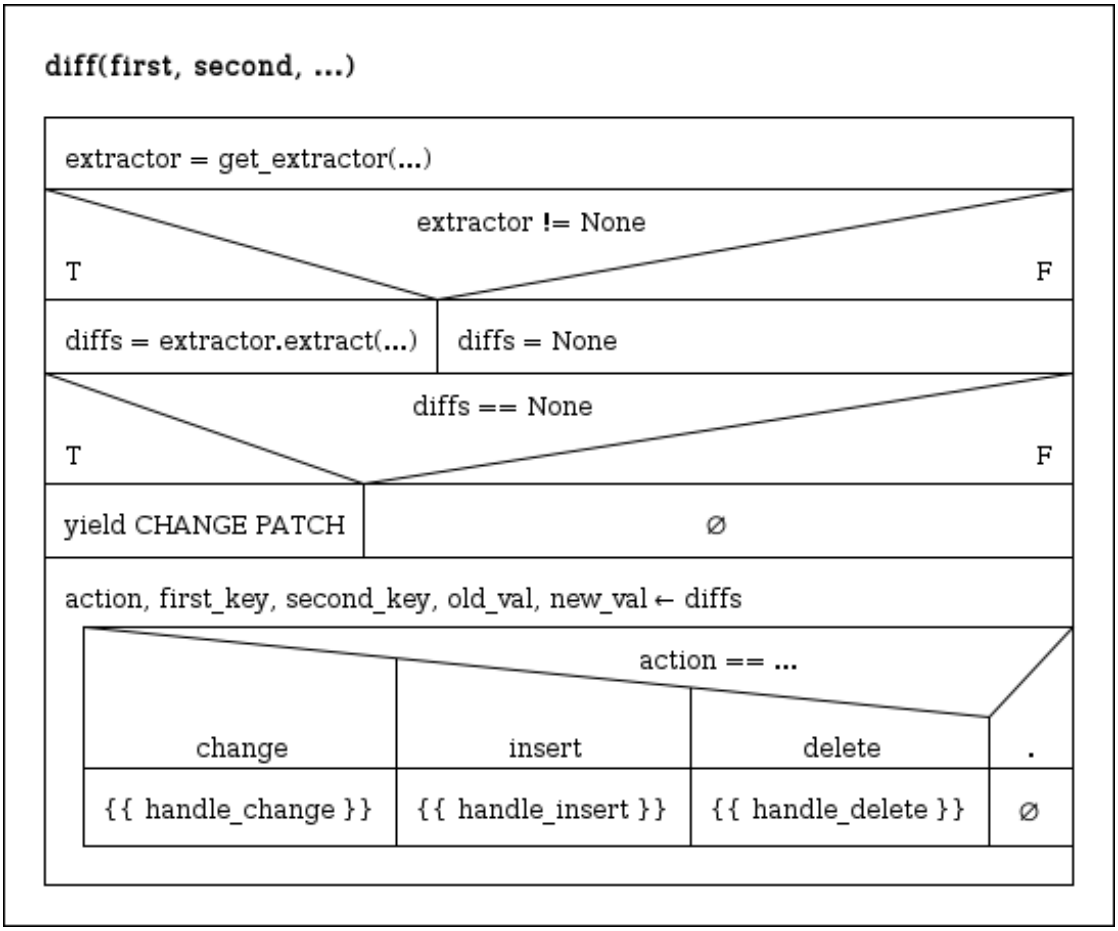

Figure 6: Structure chart of the new proposed diff function.

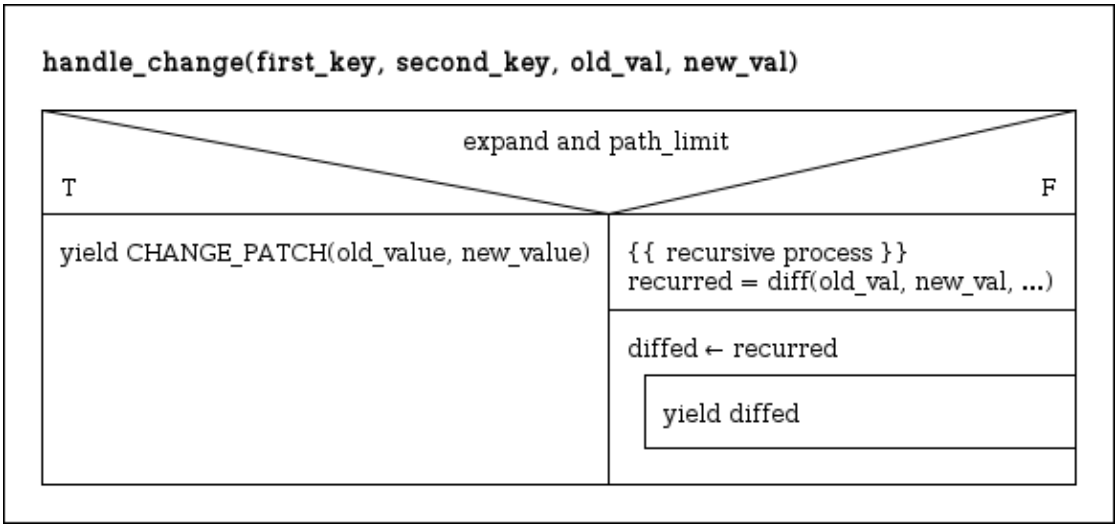

Figure 7: Structure chart of the handling of changes during the extraction process.

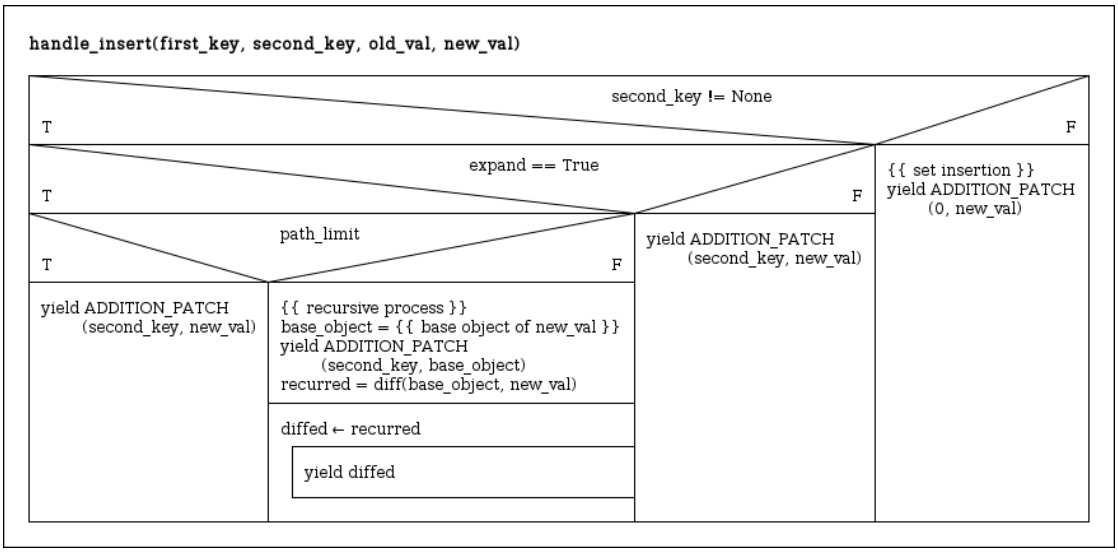

Figure 8: Structure chart of the handling of additions during the extraction process.

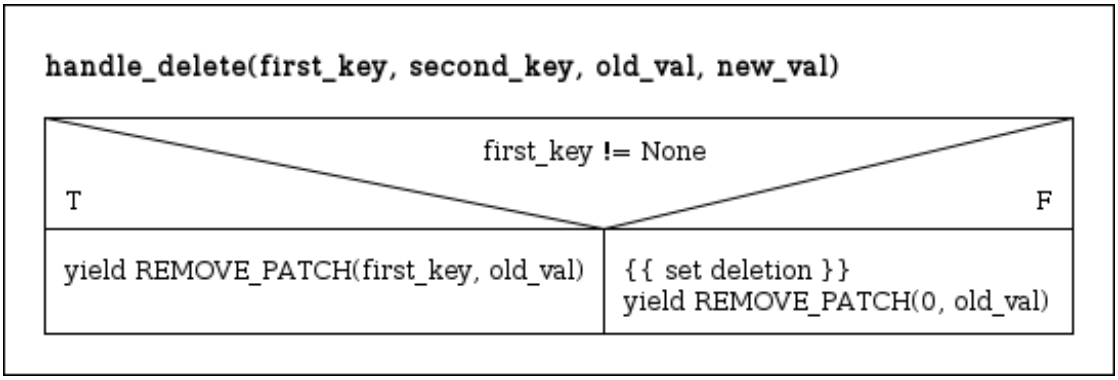

Figure 9: Structure chart of the handling of deletions during the extraction process.

One notable implementation detail is the usage of Python's 'yield' keyword. Using the yield keyword in Python basically works like a 'return' statement, with the exception that it returns a generator object.

Returning the extracted patches as a generator was mostly done due to the fact that the previous version of dictdiffer also used this feature and due to backward compatibility. Since generator objects can only be iterated once, the user might want to store them as a list by calling Python's 'list' function (see listing 30).

Listing 30: Effect of the yield keyword to the usage of the diff function.

```
list_variable = list(returns_generator_function())
```

Even tough calling the 'list' function on a list object will not change anything, it seems to be an avoidable effort.

Additionally, using the 'yield' keyword led to cleaner code, since it eliminates the programming pattern of initializing, filling and returning a list variable as shown in listing 31 and 32 [32].

Listing 31: Regular programming pattern to return a list of values

```
res = []
for x in ...:
    res.append( {{ something }} )
return res
```

It instead becomes this:

Listing 32: Returning a list of values using the 'yield' keyword

```
for x in ...:
    yield {{ something }}
```

4.3.2 Extractors

The current implementation introduces four different extractors: 'DictExtractor', 'ListExtractor', 'SetExtractor' and 'SequenceExtractor'. The first three were directly taken from the existing dictdiffer implementation and only required a few adjustments:

The old if-else statement that decided on the way of patch extraction is moved to the 'is_applicable' method and the separation into intersections, additions and deletions is basically kept the same, but substituted by using the yield keyword, since the order of extraction doesn't matter for those three extractors.

The SequenceExtractor, on the other hand, demanded a bit more effort. It utilizes Python's difflib.SequenceMatcher class to solve the 'longest common subsequence' problem, since the automated merging tool is supposed to be used in production and therefore requires robust and tested code. Additionally, since it is somehow uncertain how extensively this SequenceExtractor is going to be used, investing too much time into an own solution seems unnecessary.

According to the documentation, the SequenceMatcher class is suitable for comparing pairs of sequences as long as their elements are hashable and utilizes an algorithm, which is 'a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershelp under the hyperbolic name "gestalt pattern matching"' [34].

This "gestalt pattern matching" algorithm processes a similarity value between two, one dimensional patterns (strings). Its functionality is described in figure 10 and is sourced form [35]:

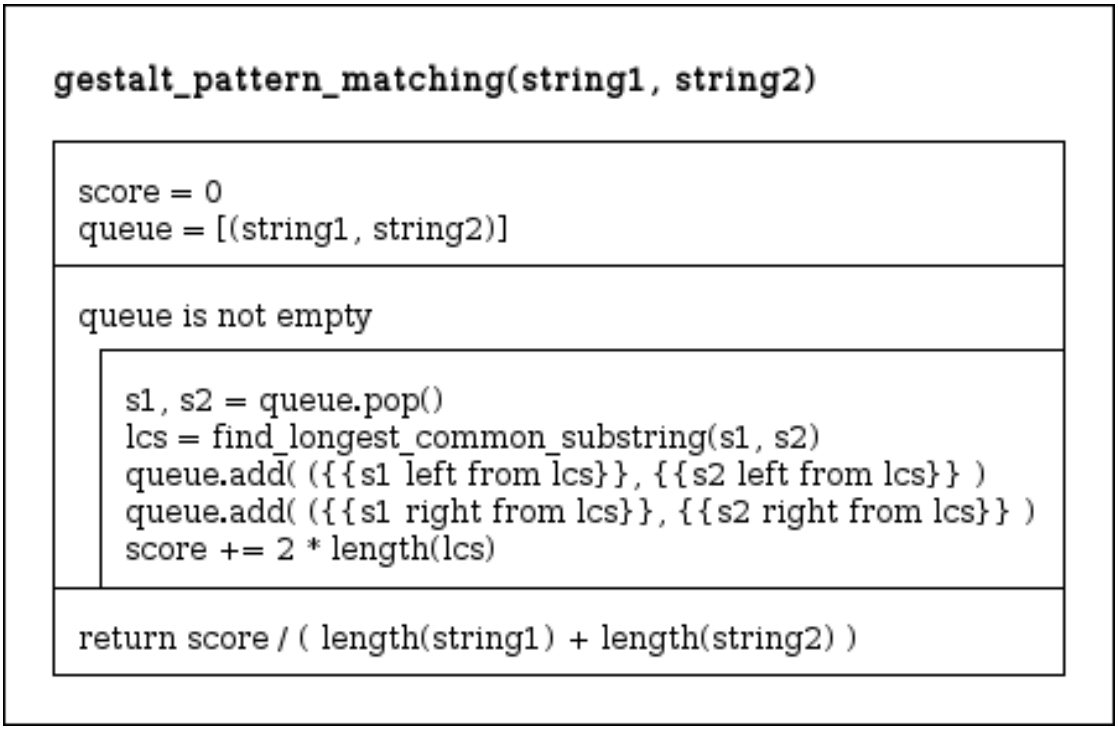

Figure 10: Gestalt pattern matching algorithm to calculate a proximity score between two strings.

The algorithm initializes a queue with the given strings, and repeats the following steps for each element from that queue:

1. Find the longest common substring of both strings.
2. Add the strings left and right from this longest common substring to the queue.
3. Add two times the length of the longest common substring to the score.

The final score is calculated by dividing the score by the combined length of both original strings.

To determine the longest common subsequence, the difflib.SequenceMatcher class alters this approach by saving the positions and lengths of the longest common substrings of each iteration. It will then examine the substrings in between those common blocks and determine the three possible options for those substrings (the following steps are assumed by reading the difflib.SequenceMatcher source code):

1. There is a substring in both original strings: Since those substrings are not part of a common substring group, they must be different, which will lead to a replacement of one string with the other.
2. There is a substring in the first original string: Since the substring is not part of the second original string, there is a deletion of this part.
3. There is a substring in the second original string: In parallel to the previous case, this means that the substring got inserted.

Case	First	Second
1	Hello, String 1 Bye.	Hello, String 2 Bye.
2	Hello, String 1 Bye.	Hello, Bye.
3	Hello, Bye.	Hello, String 2 Bye.

Table 6: Example of the three possible states for a substring while computing the longest common subsequence.

Table 6 shows a summary of the possible options for the substrings.

Once instantiated with the sequences, calling the objects 'get_opcodes' returns a list of 5-tuples containing the necessary information to transform the first sequence to the other. Listing 33 shows an extract from the docstring:

Listing 33: Extract of difflib.SequenceMatcher's man-page [34], explaining the four states of each part of the compared strings.

```
Each tuple is of the form (tag, i1, i2, j1, j2).  The first
tuple has i1 == j1 == 0, and remaining tuples have i1 == the i2
from the tuple preceding it, and likewise for j1 == the
previous j2.

The tags are strings, with these meanings:

'replace':  a[i1:i2] should be replaced by b[j1:j2]
'delete':   a[i1:i2] should be deleted.
            Note that j1==j2 in this case.
'insert':   b[j1:j2] should be inserted at a[i1:i1].
            Note that i1==i2 in this case.
'equal':    a[i1:i2] == b[j1:j2]
```

Each of those tuples don't know about the preceding ones, so every given index assumes that the sequences wasn't changed in a previous step. Due to this difference to the dictdiffer patches, which require to consider the previous patches, the SequenceExtractor class basically just counts the preceding additions and deletions and amends the indexes accordingly. Furthermore, in the case of deletions, the indexes need to be reversed to make it compliant to the old dictdiffer behavior.

One last quirk that needs to be handled originates in the SequenceMatcher behavior. In case of an 'replace' tuple from the SequenceMatcher, the presented index i1, i2 and j1, j2 can have a different range, which means that in addition to direct, index-vice changes there are also some additions or deletions. Since the dictdiffer module doesn't support actions like this, but instead handles addition, deletions and changes separately, an according transformation needs to be done. This basically just introduces two more cases:

1. The range of the indexes i1 and i2 is bigger than the range of j1 and j2: Each additional index in the i1 and i2 range will be transformed to a deletion patch.
2. The range of the indexes i1 and i2 is smaller than the range of j1 and j2: Each additional index in the j1 and j2 range will be transformed to an addition patch.

Due to the fact that the extraction process doesn't require to store additional information during multiple calls of the same method, the funcitonality is implementated as class methods.

It should be noted that the information of the original position of the change, which is given by the SequenceMatcher class, is lost at this point and needs to be restored in a later step.

4.4 Conception of the automated merging tool

After describing the amendments to the dictdiffer module in order to improve the difference extraction process for the automated merging process, the following sections are going to be about the conception of the actual automated merging tool.

4.4.1 The rule system

The provided merging functionality of the automated merging tool should be capable of working with as little human interaction as possible (meaning the case of a failed automated conflict resolution). The general idea being, that the merging tool follows some sort of 'rules', which map certain conflicting changes in the data structure to corresponding actions, which will try to resolve the conflict.

There are certainly multiple different solutions to a rule based system like this, but the following two options seem to be the most promising.

The programmatic approach
This would be a simple mapping of a certain path in the data structure to a function, that handles the given element. This approach would probably require two persons with different backgrounds. One would be needed to define the actions necessary to solve the expected problem for a given path and the other would needed to realize those demands.

A domain specific language
A 'domain specific language'[17] would mean a defined language for the domain of handling conflicts. This would also include a mapping between a path in the data structure to the function, but this function would be written in the domain specific language, which might make it possible for persons without a programming background to define and realize the desired behavior and solution to the given problem.

Designing a custom domain specific language has been deemed the less promising approach, because of the following reasons:

- The scope of possible future tasks is unpredictable. Every other area of application where an automated merging tool can be used would lead to another domain specific language or to an extension of the existing one, which might then end up as a fully featured programming language.

[17] "small languages, focused on a particular aspect of a software system"[38]

- The scope of the record merging domain is already hard to oversee. Special combinations of repositories and record types probably need very specific actions, which are hard to predict.

The unpredictability points to a domain specific language that should handle every possibility, which would lead to designing a 'general purpose'[18] language.

Additionally, choosing the programming approach would allow to integrate something like a configuration file parser, which could parse a later introduced domain specific language.

So the programmatic approach is the one with the most possible use cases and is therefore chosen in this thesis.

The next topic to decide upon is how the mapping is going to be handled. From the recent knowledge of the record merging process, it seems that there is the following distribution of cases: The majority of fields will be handled in the same way: Take the newest update and do not overwrite human made changes. Whereas the remaining minority of fields need very specific actions. One example, which will be further discussed in section 7, are the fields containing the references and authors.

It would be cumbersome for the user to map every possible field of a record to the same function, so there need to be some kind of wildcard mechanism, which maps more specific paths to the same function. This approach advances to a cascading approach, where a given path is mapped to the most matching function first, and then moving down the hierarchy.

4.4.2 General requirements

The process of merging two data structures while using the patches extracted from invenio.dictdiffer requires multiple, distinct steps which might amend the data. This basically means that the merging process is a sequences of states. Since functional programming tries to avoid states outside of function parameters[31], those parameter lists would become extensively long. It becomes also apparent, that in case of a failed merge a lot of those states need to be recovered, leading to big tuples of returned values. It might also be more difficult for the user to distinguish which function to call on which object.

Due to those reasons, the automated merging tool will be designed in an object oriented way. Gathering all the relevant steps for a successful merge in one wrapper object, which executes the rest.

[18] something in the scope of Python

Every distinct step itself is then also realized as an object, providing a specific interface.

This holds multiple benefits beside those provided by the object oriented approach in general, like logical grouping of functionality:
Together with Python's ducktyping[33] and dynamic parameter assignment, it is possible to exchange the different parts of the merge-object rather quickly and in multiple ways. For example by subclassing or simply by reassigning certain methods or attributes of objects. It furthermore allows to introduce some aids for possible speedups.

4.4.3 General structure

After extracting the patches, the next step is to merge them into one list of none conflicting patches, which would, when applied using dictdiffer's patch function, transform one data structure to the desired outcome. To do so, three things need to happen:

1. Recognize conflicts.
2. Resolve those conflicts.
3. Unify the patches to one.

The 'Recognize conflicts' step needs one additional preparation step, since the dictdiffer.diff function loses certain information, which need to be regained. To achieve this, a wrapping step is introduced, which is linked to the way the patches were extracted and tries to regain the information lost in this process.

Overall, this leads to the following flow of events for the merging object:

- Extract the patches for the data structures at hand.
- Wrap those patches to regain the needed information.
- Find conflicts.
- Resolve those conflicts.
- Unify the patches to build one coherent list of patches.

The following sections will introduce the concepts of each of those steps, starting at the wrapping step, since the patch extraction was explained in the section 4.1.2.

4.4.4 Wrap

The Wrapper class has to fulfill two tasks: Firstly, it needs to wrap the given patches in an object that allows easier handling for the following steps of the merging process.

Secondly, it needs to regain information that was lost during the extraction process.

So in general, the wrap function simply iterates over the extracted patches and puts them in the structure displayed in figure 11:

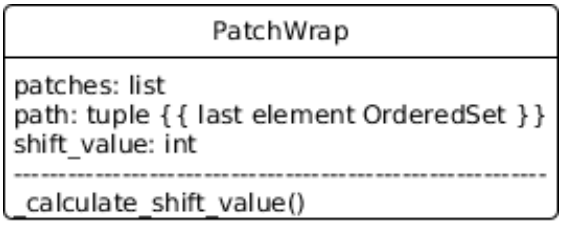

Figure 11: Proposed PatchWrap class to group patches and to provide the original path.

In the most general case, the wrapper will just put one patch in the patches list and the 'path' key of the wrapped element will be the same as the path of the patch.

In the more special cases of patches extracted using difflib.SequenceExtractor, two important information are lost:

- Which patches build a semantic group (as far as this can be extracted correctly)
- What was the original place in the sequence to make the changes at hand

Depending on the use case, those two bits of information can be important. Having an incomplete conflict can be as bad as having false-positive conflict due to a wrongly extracted path.

Unfortunately, the extracted patches do not contain any information helping to decide which patches are grouped or not, so specific <NAME>Wrapper classes that decide which patches need to be grouped are needed.

To make this happen, there needs to be a close link between the way patches are wrapped and how and where they got extracted. This leads to a few similarities: The corresponding wrapper will be chosen based on the type of the data structure, the path of the patch at hand or in an 'is_applicable' way, like the extractors.

Contrary to the extractors, the wrappers need to store some additional information like previous additions and deletions, which leads to the fact that they will need to be instantiated objects, in contrast to a collection of class functions in case of the Extractor classes.

4.4.5 Conflicts

Once the patches are grouped and wrapped, conflicting patches need to be assigned to each other.

To decide if two patches are conflicting, paths and actions are required. Those are the possible scenarios for conflicts:

- The path of the patch is the same.
  - This is always a conflict, except if a addition and a change patch have the same path in a list, that is treated like a sequence. In this case, the addition patch inserts the change before the assigned path.
- One path is the super-path of the other. This is, due to the extraction process, only a conflict if the super path patch is a deletion patch.

So the ConflictFinder class will compare each wrapped patch from one data structure to each wrapped patch of the other, runs a 'is_conflict' method and stores the results in a list of Conflict (shown in figure 12) objects.

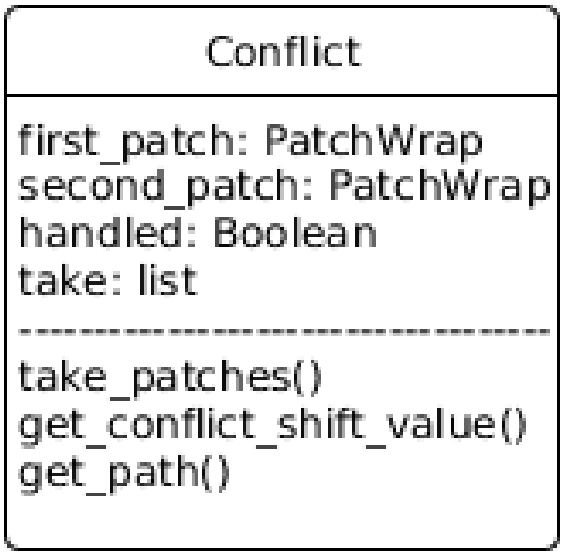

Figure 12: Proposed Conflict class, storing two conflicting PatchWrap objects.

4.4.6 Resolve

The Resolver class iterates over the previously determined conflicts and presents them, based on the path, to an resolve function. In case that this function is not successful, the conflict is stored in a list of unresolved conflicts, which will be returned as an exception for the user to manage.

At this point the previously answered question of how to manage the rules leads to the following addition to the simple resolution loop: In case that one function could not immediately resolve the conflict, the path should be stripped by the last element to find a more general rule. This, of course, might not always be desired, so a special exception needs to be introduced to stop this process.

In order to provide as much information and possibilities, those resolve functions receive the conflicting patches, all other patches and any other additional information as parameters.

Since conflicts are determined based on their path and action, patches might be conflicting when in fact they are not. This includes patches that perform the exact same changes. To reduce some work, the resolver should also try to auto resolve those conflicting patches.

To later on decide which patches should be taken, the wrapped patches 'take' field, which is a list of indexes stating which of the patches should be used later on, will be amended.

This 'take' approach can lead to inconsistencies. Since the path of every patch assumes that all the previously extracted patches are applied, skipping patches can lead to index out of range errors or wrong positions in the sequence. The responsibility to avoid this is currently left to the user.

4.4.7 Unify

The Unifier class produces a single, consistent list of invenio.dictdiffer patches by iterating over all wrapped patches and picking those, which are returned by the 'take_patches' function.

The following cases need to be handled by the Unifier class:

- The wrapped patch is not part of a conflict, which means that every contained patch will be put into the unified list.
- The wrapped patch is part of a conflict, which leads to calling the 'take_patches' method to receive the necessary patches.

- The wrapped patch is part of a conflict, but was already handled, which leads to skipping the patch.

There is one additional major problem that needs to be solved by the Unifier class. Unifying patches in a sequence translates to indexes, which need to be shifted according to their predecessors.

Even simple examples of non conflicting additions to a list in a alternating fashion leads to shifts, since the original extraction process doesn't and can't care about possible other additions coming from a different source.

Listing 34: Illustration of the necessity of shifting patches during the unification process.

```
original = [0,     2,     4]
source1  = [0, 1, 2,     4]
source2  = [0,     2, 3, 4]

Patches original -> source1:
('add', '', [(1, 1)])

Patches original -> source2:
('add', '', [(2, 3)])

Correct merge:
merge    = [0, 1, 2, 3, 4]

Merge using unchanged indexes:
merge    = [0, 1, 3, 2, 4]
```

The listing 34 shows a simple example of the problem. Inserting one additional element at an non-conflicting index leads to wrong indexes for every following patch from the other source. To react to this, the additions and deletions need to be counted, and the patches need to be changed accordingly.

4.5 Realization of the automated merging tool

This section will deal with the realization process of the previously discussed structure of the automated merging tool. It's structure will follow the steps necessary for a successful merge and will then describe the Merger class, which will wrap it all together in one interface.

4.5.1 Wrapper

The Wrapper class itself handles the necessary management needed by the merging process to prepare and select the necessary <NAME>Wrapper classes. It therefore provides the two methods shown in listing 35:

Listing 35: Methods provided by the Wrapper class

```
def wrap(self, patches, _source1, _source2)
def update_wrappers(self, update)
```

The 'wrap' method iterates over the presented patches and decides on the appropriate wrapper for each patch. Unfortunately, there are a few quirks to this process:

1. The decision process should be equal to the extraction process, meaning that the required wrapper should be decided upon using the same kind of information, the type of the node in the data structure, the path or a general default option. At this point, the already extracted patches are the main concern, so in order to acquire a wrapper by type, the path needs to be extracted from the patch, and then the data structure needs to be traversed.

2. Since the patches don't know anything about their predecessors or successors, every patch needs to be checked for a possible grouping. In the current implementation, the wrapper would do some foreshadowing which is unknown by the following patches. The wrapper objects would therefore need to keep track of already checked patches, returning a None value in case that the patch is already part of a wrapping. This behavior leads to the necessity for the Wrapper class to perform a None check before appending the result of each 'wrap' call before appending it to the result list.

The 'update_wrappers' method is a utility function which serves the necessary connection between the extraction and the wrapping process. Starting from the dictionary structure that contains the keys to choose the right extractor for the current situation, the 'update_wrappers' method iterates over this dictionary and instantiates the corresponding wrapper object for the given extractor. This way it is ensured that the right wrapper object is chosen.

In the current state of the automated merging tool, there are two individual wrapper classes, 'DefaultWrapper' and 'SequenceWrapper'. The DefaultWrapper is responsible for the wrapping process of those patches that were extracted without considering the semantic meaning of their position in the data structure. The SequenceWrapper corresponds to the SequenceExtractor. Both classes descend from the BaseWrapper class, which provides the general functionality of extracting the path from a given dictdiffer style patch.

The DefaultWrappers 'is_applicable' method always returns True and the 'wrap' method simply wrap the present patch in a PatchWrap object, since it doesn't consider any succeeding patches.

The SequenceWrapper, on the other hand, has to rebuild the information of semantic grouping and the initial position of the changes. It therefore needs to keep track of the following: Additions made, deletions made and the current super path.

For each patch given to the 'wrap' method, it will execute the steps described by the figures 13, 14, 15 and 16 :

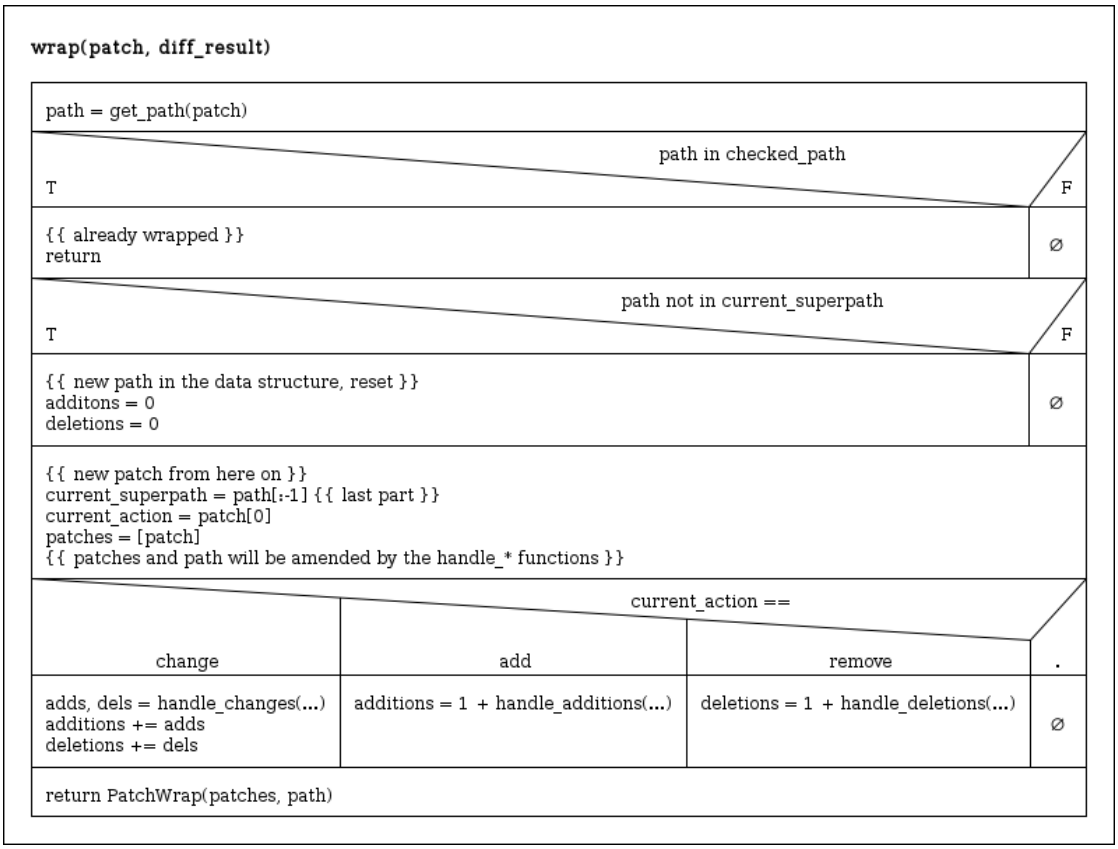

Figure 13: Wrapper class' wrap function structure chart.

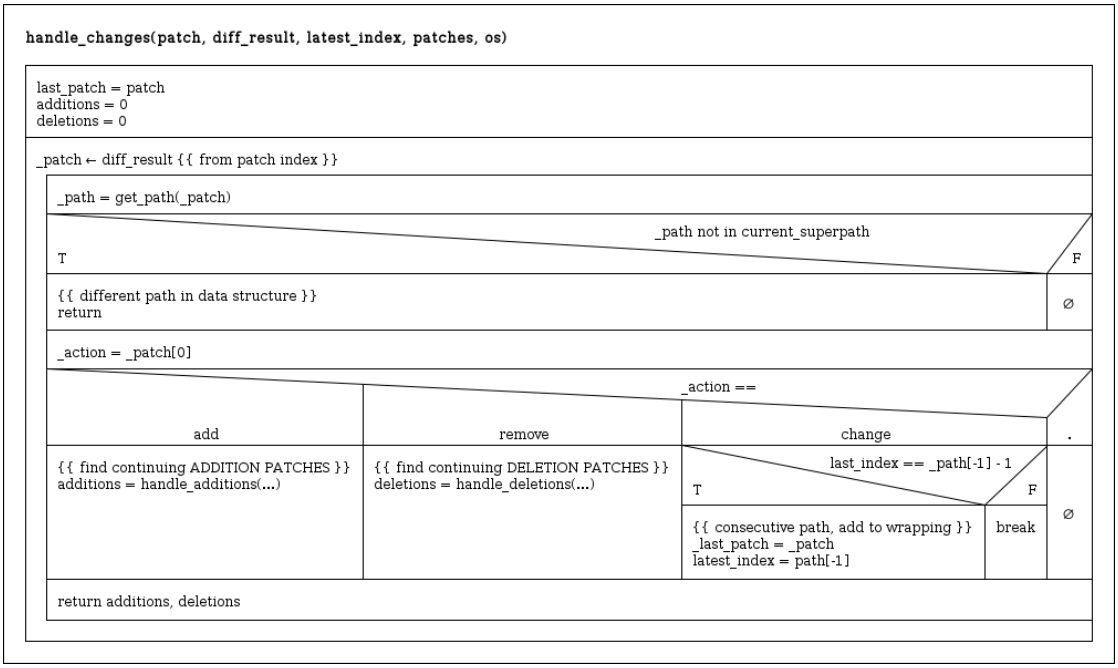

Figure 14: Structure chart of the handling of change patches during the wrap process.

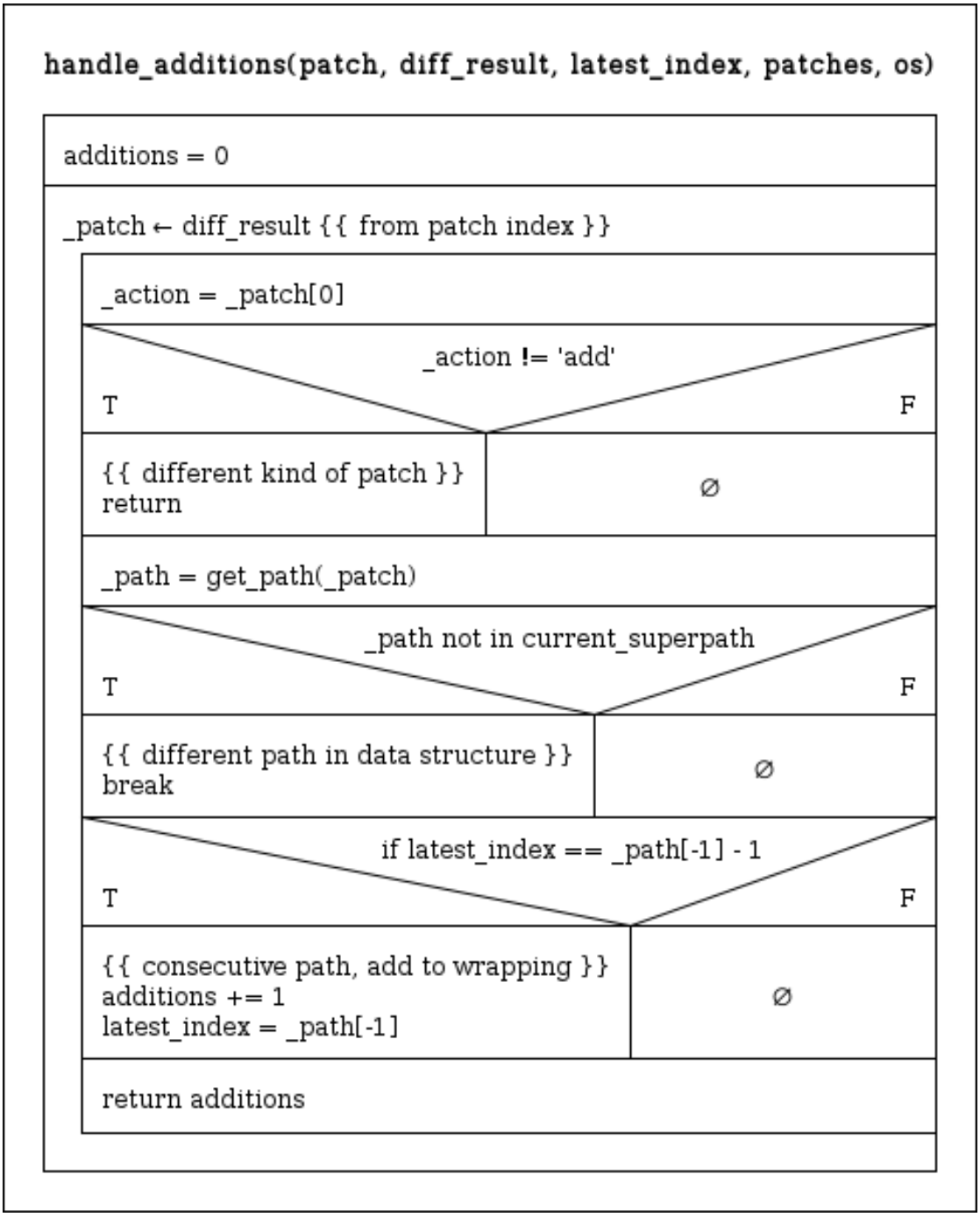

Figure 15: Structure chart of the handling of addition patches during the wrap process.

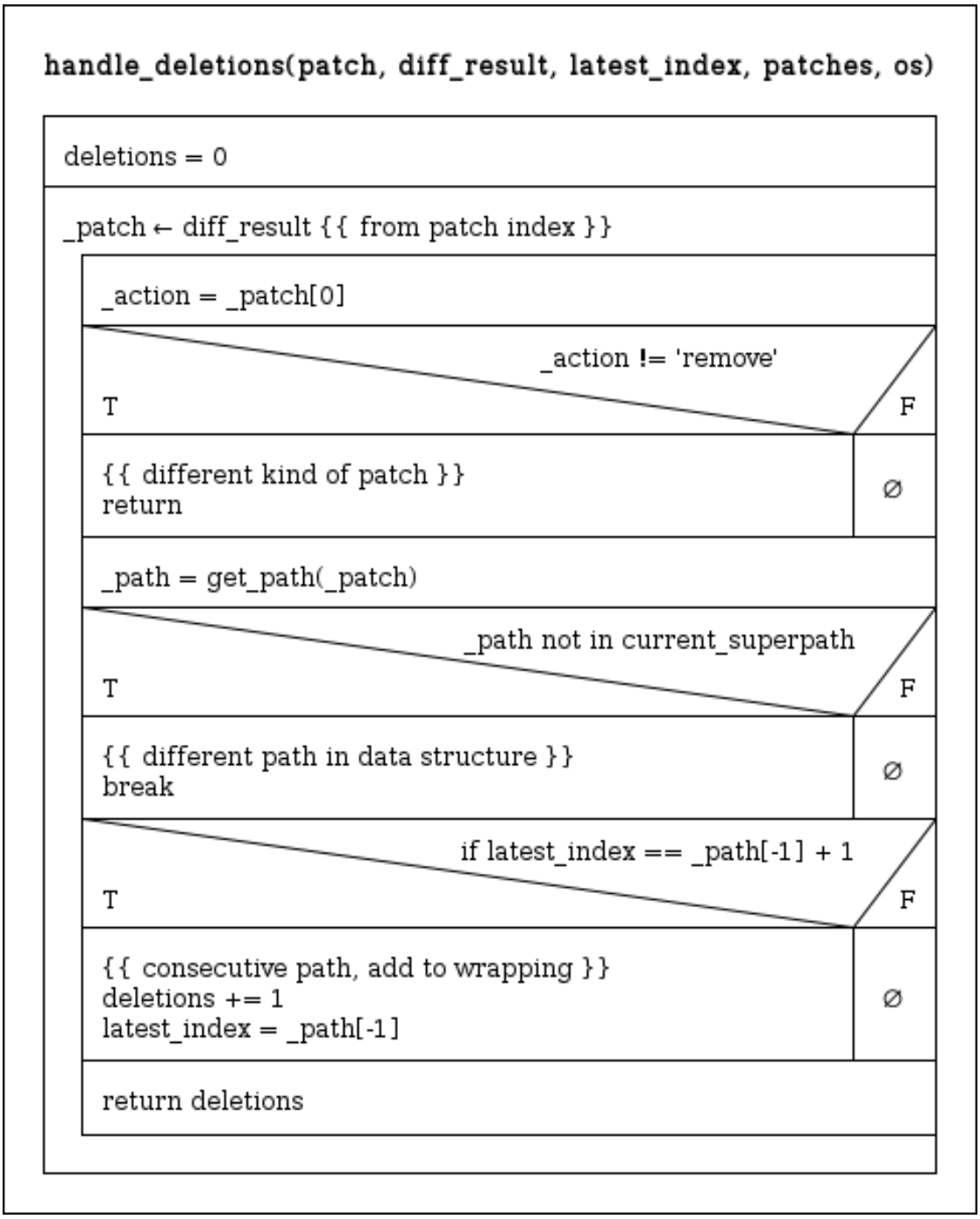

Figure 16: Structure chart of the handling of deletion patches during the wrap process.

At first the 'wrap' method checks if the patch was already part of another wrap, skipping it in that case. If the patch is indeed new, the current action will be recognized as well as the current super path and the current index in that super path.

Starting from the current action variable, three possible situations need to be handled:

1. In case of an 'add' or 'remove' action and considering the extraction process using the difflib.SequenceMatcher class, it is certain that only patches of the same kind with an ongoing index can follow (consecutive deletions have decreasing indexes). This makes it easier to check the succeeding patch for their affiliation to the current patch.
2. The remaining kind of patch had three different forms:
   (a) Only 'changes'.
   (b) Changes followed by additions. In this case, the index of subsequent patches will be a sequence without skips, which makes it as easy to detect as regular 'change' only, 'add' or 'remove' sequences.
   (c) Changes followed by deletions. This case is unfortunately tricky, since deletion patches have a reverted index, there will be a jump between the indexes of the 'change' patches and the deletions. To still recognize those cases the SequenceWrapper needs to traverse the patches until he find a sequence of deletions, which end with the next index of the preceding changes sequence.

In addition to this, the SequenceWrapper also needs to regain the information of the index where the change appeared originally. It therefore counts the additions and deletions and, considering the type of sequence, adds and subtracts them accordingly.

There are five different kinds of patch sequences:

1. Additions only: This will only contain the index of the initial patch.
2. Deletions only: This will contain every index of the grouped patches.
3. Changes only: This will contain every index of the grouped patches.
4. Changes plus additions: This will contain every index of the changes patches of the group.
5. Changes plus deletions: This will contain every index of the grouped patches.

4.5.2 ConflictFinder

The ConflictFinder class simply iterates over every combination of patches extracted from the latest common ancestor and the first source and the patches extracted from the LCA and the second source and run the 'is_conflict' method on them.

This method checks for the in section 4.4.5 described conditions for a valid conflict and wraps the patches in a Conflict (12) object.

The Conflict class implements the methods 'take_patches', 'get_conflict_shift_value' and 'get_path', which provide functionality necessary in later step of the merging process.

The 'take_patches' method returns those patches, which are marked to be taken. It therefore iterates over the 'take' attribute of the class, which contains a list of the kind shown in listing 36.

Listing 36: Illustration of the take attribute of the Conflict class.

```
take = [({{ first or second patch }}, {{ index }}), ...]
```

This structure makes it easy to choose, mix and leave out certain patches, in case that ever becomes necessary. For each tuple in the list, the method yields the patch found in the corresponding attribute storing the patch and the indicated index.

The 'get_conflict_shift_value' method uses the same 'take' member to calculate the values required to shift the succeeding patches in the according patches lists. It therefore considers the supposed shift value in case that every patch would have been applied, and subtracts and adds the real values.

Listing 37: get_conflict_shift_value example

```
pw1 = PatchWrap([('add', 0, ['foo', 'bar']),
                 ('add', 1, ['apple', 'banana'])])
pw2 = PatchWrap([('add', 0, ['apple', 'mango'])])

c = Conflict(pw1, pw2)

The original shift value for pw1 is 2, while the original
value for pw2 is 1.

Take every patch from pw1:
c.get_conflict_shift_value() -> 0, 1
Since there will be one more patch than originally, the
patches in the list handled by pw2 need to be shifted by +1.

Take every patch from pw2:
c.get_conflict_shift_value() -> -1, 0
Since there will be one patch less than originally, the
patches in the list handled by pw1 need to be shifted by -1.
```

The 'get_path' method is a simple utility method for the case that the conflicting patches do not have the same super path, which can occur in the case of a deletion.

4.5.3 Resolver

The Resolver class offers the user an interface in form of the two functions displayed in listing 38:

Listing 38: Methods provided by the Resolver class.

```
def resolve_conflicts(self, conflicts)
def manual_resolve_conflicts(self, picks)
```

The Resolver class get initialized given the patches, the provided actions and 'additional_information' object. After initialization, the next step is to call the 'resolve_conflicts' method, which is described in figure 17.

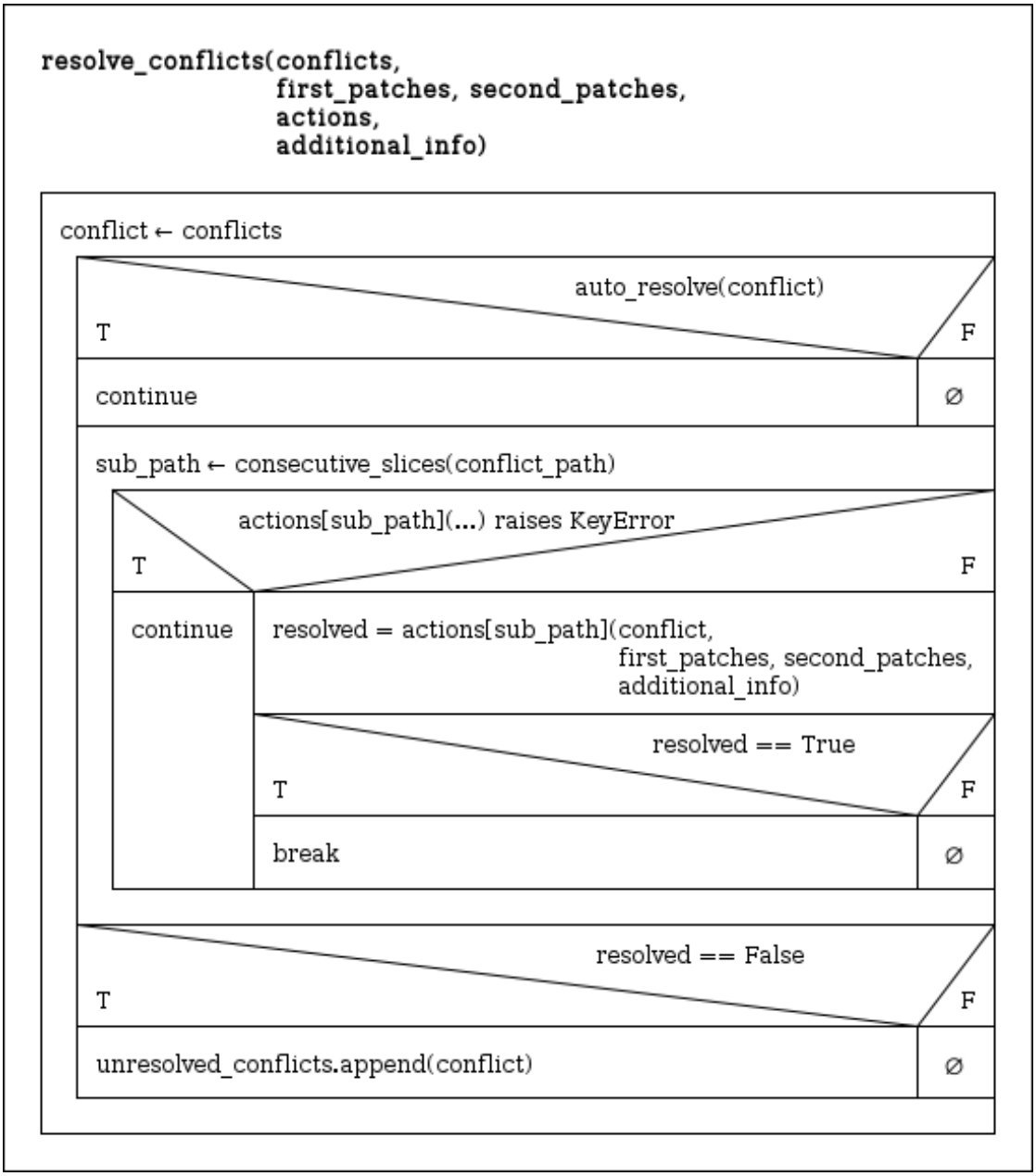

Figure 17: Structure chart of the resolge_conflicts method of the Resolver class.

It iterates over the conflicts, finds the conflicting path, which in the current implementation returns the shortest path in order to address conflicts involving deletions, and tries

to resolve the conflict in case the patches simply do the same.

If this auto resolve fails, the provided actions are utilized. According to the resolution conception, the conflicts are presented, based on the given path, to an resolve function, which is stored in the actions object.

In order to make this process easier, the WildcardDict class was introduced, which utilizes the characters '*' and '+' to indicate wildcards. The '*' character allows every possible sequence of keys following the same predecessors as the '*' to be mapped, whereas the '+' character only allows every key in this position. Some examples of valid and invalid keys are shown in listing 39.

Listing 39: Matching behavior of the WildcardDict class.

```
('authors', '*')
    ... would match for:
    ('authors', 1)
    ('authors', 2, 'name')
    ('authors', 3, ...)
    ... but not for:
    ('authors',)
    ('foo', 1)
    ('foo', 'bar')

('foo' '+')
    ... would match for:
    ('foo', 'bar')
    ('foo', 1)
    ... but not for:
    ('foo',)
    ('foo', 1, 'name')
    ('bar', 'foo')
```

In case that a conflict can not be resolved, it will be stored in a 'unresolved_conflicts' variable, which is then returned throwing a UnresolvedConflictsException exception.

The 'manual_resolve_conflicts' method is supposed to be called to solve the unresolved conflicts and it directly utilizes the conflicts 'take_patches' mechanics, leading to the 'picks' parameter being a list of lists, which will be passed to the conflicts 'take' member. Figure 18 shows the process.

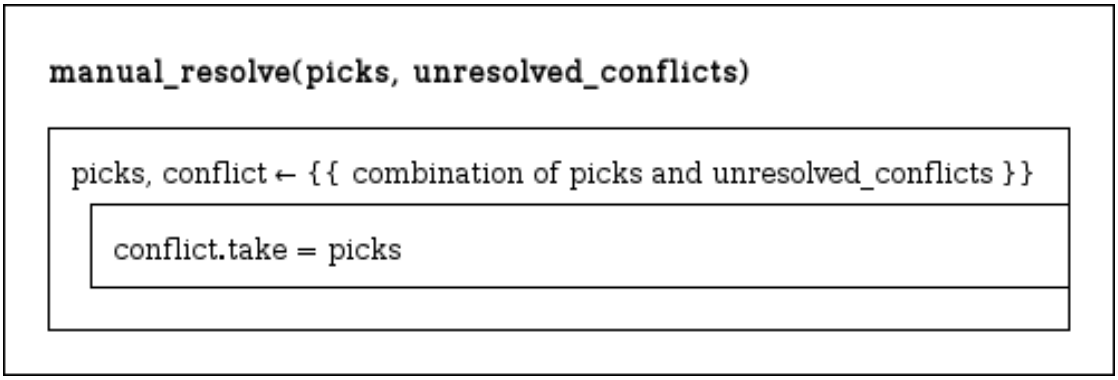

Figure 18: Structure chart of the manual_resolve function.

4.5.4 Unifier

The Unifier class solves the issue of combining the list of meanwhile merged patches.

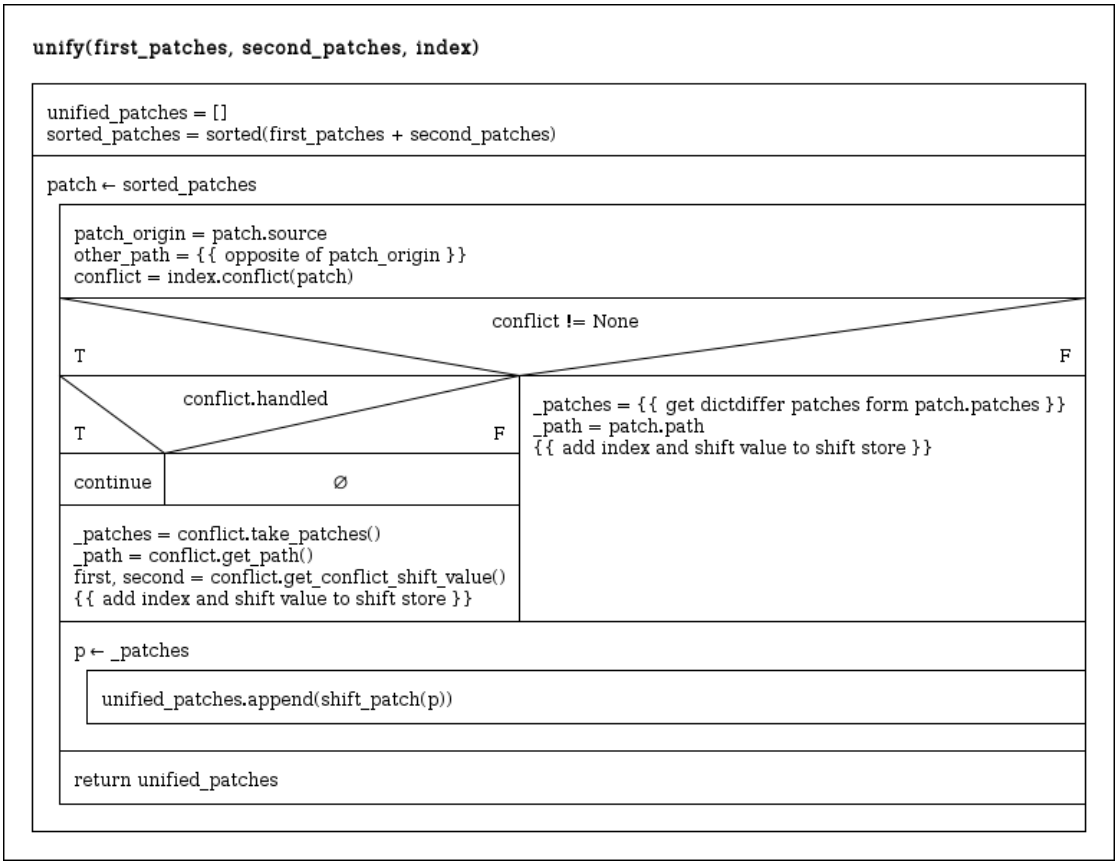

Figure 19: Structure chart showing the Unifier unify method.

The Unifier class, as shown in figure 19 iterates over every patch, which are stored in a sorted list of the combination of the two patch lists. This sorting is necessary due to the way the Unifier class keeps track of the necessary shifts. As illustrated in the concept section, to unify patches, their indexes need to be shifted, for this shifting, it is necessary to know from which index we would have to start shifting and by how much.

The Unifier class utilizes a variable called '_shift_store' which is a dictionary object, that keeps track of the shifts based on their origin and the current super path. For each patch the class will look up its origin, its super path and its current index, looking up the respective position in the '_shift_store' to shift the path of the patch.

4.5.5 Merger

The Merger class is a simple utility class responsible for wrapping all the previously described functionality. It provides the interface of two methods, 'run' and 'continue_run'.

The former method starts the whole merging process with tow possible outcomes, everything worked out and it stores the unified patches, or something couldn't be resolved and it raises and 'UnresolvedConflictsException'. The latter cases requires a human to decide which patches to pick from the given unresolved conflicts.

The second method, 'continue_run', continues the resolution process with those picks and then continues the automated merging process.

4.6 Theoretical evaluation of the automated merging tool

Evaluating the applicability of the realized automated merging tool is an important aspect of the thesis, which proves to be difficult.

One approach would be to test the possibilities against real life examples, where the file, the update and the desired merge are already known, since it was done by an operator already. This procedure is problematic in the sense that finding useful test cases of this sort is difficult and time consuming. Additionally, providing every resolution function to solve the given conflicts would, in most cases, not provided a lot of value for the evaluation process, but take time to implement.

So this thesis will try to consider the possible and desirable amendments that the automated merging tool would have to be capable to perform from a theoretical point of view and show if the tool could handle the demand or not.

There are four basic operations for a data structure: create, read, update and delete (CRUD). Starting from those four operations, neglecting the read operation since it doesn't demand any amendment to the data structure, this section is going to consider the demands for every operation.

The dictdiffer patches can amendment the handled data structures in every possible way (it should be noted that it is not possible to transform the root object, for example from a dictionary to a list, but this is not a scenario relevant to the bibliographic metadata environment), which is given just utilizing the deletion and addition patches: A delete patch can simply delete an element at a given position while an addition patch can insert everything into the data structure.

Therefore, it becomes apparent that the resolution actions can amendment the data structures in every desirable way, since they have full access to the complete list of patches

as well as to the data structures before the patches are applied. They could therefore, in the most extreme case, just replace the complete list of patches with their own amendments, deleting all the content of the latest common ancestor before adding the desired content.

5 Config Manager

The config manager is supposed to be an utility class that can provide all the previously described configurations for the patch extraction process from a configuration file.

So this section will be divided the following way:

- Requirements: Which configurations of the patch extracting process should be configurable?
- Configuration file design: Which design of the configuration file would be most suitable?
- Realization: Description of the realization process

5.1 Requirements

The automated patch extraction process in general needs one piece of information: Which path of the data structure needs to be handled by which function, so the first obvious concept of a configuration file is exactly that, a mapping from a path to an action as illustrated in listing 40:

Listing 40: Initial thought regarding a configuration file schema.

```
<path>      =   <action>
```

Unfortunatley, the patch extraction needs some configuration so the extraction is neither too shallow, nor too deep. It would require the user to provide an unreasonable amount of actions to handle all the possible patches. Listing 41 explains the problem with an example.

Listing 41: Different extraction results utilizing the path limit functionality.

```
LCA
{'author': {'name': 'Jo'}}

NEW
{'author': {'name': 'Bob'}}

Patches (default)
('change', 'author.name', ('Jo', 'Bob'))

Patches ('author' as path limit)
('change', 'author', ({'name': 'Jo'}, {'name': 'Bob'}))
```

Fortunately, the desired path limit is already existing in the form of the paths in the configuration file. So by setting the extraction process to 'expand' and using the path limits based on the configuration file paths, the patch extraction becomes predictable.

Furthermore, it allows to use the configuration file in a cascading way: super paths of certain data structures don't have to deal with the same problem anymore because of the extraction process, but can handle more general problems.

Listing 42 shows the outline of a configuration file utilizing cascading actions using the wildcard notion provided by the WildcardDict class.

Listing 42: General outline of the configuration file handled by the ConfigManager class.

```
<path>        =   <action>
foo, bar      =    solve_specific_foo_bar
foo, *        =    Solve_general_the_rest
```

Additionally, it might be necessary to filter or adjust patches before trying to solve them. Possible scenarios would be the amendment of a string format or the simple case of dropping certain patches because the affected field is of no interest. It should be noted, that the realization of this part is not part of the thesis, it is just considered as a future step in the merging process.

The general approach is very similar to the resolving configuration: It is a mapping from a path to an action, but there is one difference: In case of a resolving actions, there is no way that they could affect just the patches from record one but not those from record two, in the case of filtering patches, this might be the case.

There seem to be two possible options to solve this:

- Instead of a direct mapping, introduce another column in the configuration file, so there is a mapping from a path to an action for the patches of record one and for the patches in record two
- Allow the mapped action to be a list of maximum two actions. The first one would then decide if it is applicable.

Unfortunately, there doesn't seem to be a wrong or right decision here. Adding a third column here appears to be less user friendly, since it would introduce redundancy or a special character to skip a source.

There are two possible cases that need to be handled by the configuration in this case. The action applies to both records, or only to one of them.

Applying the action to both could be done by explicitly stating that in the column, or by assuming that if one column is empty, it applies to both (see listing 43).

Listing 43: Possible configuration file schema, utilizing tabulators to ditinguish the application of an action to a source.

```
<path>      =   <source1>   <source2>
foo, bar    =    action      action

foo, bar    =    action
```

The first option would be pretty redundant, but maybe easy to read. The second option introduces the following problem: What happens the user wants to skip a source? It would be necessary to add a special character. Just adding a tab or a whitespace can be fine for 'source1', but becomes not recognizable for skipping 'source2' (see listing 44).

Listing 44: Approach of skipping the application of an action to a source using tabulators.

```
<path>      =   <source1>   <source2>
foo, bar    =                action      #skipping source one

foo, bar    =    action                  #skipping source two?
```

Utilizing a special character as shown in listing 45 would work:

Listing 45: Approach of skipping the application of an action to a source using a special character.

```
<path>      =   <source1>   <source2>
foo, bar    =    ;          action      #skipping source one

foo, bar    =    action     ;           #skipping source two?
```

The alternative, introducing a possible second action as shown in listing 46, might look cleaner:

Listing 46: Second approach to assigning actions to sources

```
<path>      =   <action>
foo, bar    =    action              #applying to both

foo, bar    =    skip_source1 action #skipping source one
foo, bar    =    skip_source2 action #skipping source two
```

So in this thesis, the second option is proposed.

5.2 Configuration File

The previous thoughts accumulate in the configuration file template of listing 47:

Listing 47: Final configuration file template

```
[FILTERING]
<path>        =    <deciding_action>? <action>

[CONFLICTS]
<path>        =    <action>
```

Using a configuration file layout that is similiar to Microsoft Windows INI files might be reasonable in the sense of relying on a proved concept as well as it enables the use of Python ConfigParser[19], which will simplify reading the files.

5.3 Realization

The ConfigManager class utilizes Python's ConfigParser module, namely the ConfigParser class, which allows to easily parse Microsoft Windows INI style configuration files[39].

Given this start, the ConfigManager class will then load then Python file containing the action corresponding to the configuration file. The default behavior here is to look for a file with the same name as the configuration file.

The ConfigParser.ConfigParser class stores each line following a section header and makes them accessible using the 'items' methods, this allows the ConfigManager to iterate over the contents of the two parts, FILTERING and CONFLICTS, of the configuration file individually, extracting 'key_limits' and actions mapped to the defined paths from the given information.

At this point of the implementation, it is possible to provide a 'deciding_action' previous to the real action, which is handled by the utility method '_prepare_action'. Unfortunately, this feels like a cheap re-implementation of Python's annotation, so it is likely going to be a rather useless feature.

[19] a configuration file parser for Microsoft Windows INI file like files[39]

6 Integration

This section is going to describe the integration of the realized automated merging tool into the existing Invenio environment. At the time of this writing, the software is supposed to be integrated into Invenio v2 using the existing Workflows and Holdingpen module.

The integration will be considered as a functional proof of concept, therefore not dealing a lot with implementation details and UX/UI standards. It is just supposed to show one possible way to integrate the automated merging tool into the existing environments.

This section will be divided into the following parts: The first deals with the Invenio.Workflow module, what it is and how the automated merging tool is integrated into it. The second part addresses the same aspects of the Holdingpen interface (technically part of the Workflow module), which allows user interactions for Workflow tasks.

If not referenced otherwise, the information displayed in this section was gathered through conversations with Jan Age Lavik, developer at CERN responsible for the Workflows and Holdingpen module (last conversation: 20.02.2015).

6.1 Workflows

The Workflows module contains a so called Workflow engine, which is a finite state machine including persistence [40]. The Workflow engine drives a Workflow object through a sequence of tasks and can be stoppend and continued at any given point of that sequence [40]. Workflows is therefore a use case for processes, which need to executed on a regular basis or for multiple data objects.

The additional features provided by Workflows are there for convenience. Processes that might fail and demand human interaction at certain points can be stopped during their task sequence for the required alterations and continued afterwards, without the necessity of rerunning every step from the beginning.

Due to those reasons, integrating the automated record merging as an Workflow is very convenient. It can run on a scheduled basis, mostly autonomous, and can be supervised by a human being if necessary.

6.1.1 Details and usage

A Workflow is defined as a sequence of functions, which are called in order. Listing 48 shows an example implementation of a Workflow.

Listing 48: A Workflow example, showing its outline as nested, consecutive function calls [40].

```
[
    check_token_is_wanted, # (run always)
    [                      # (run conditionally)
        check_token_numeric,
        translate_numeric,
        next_token         # (stop processing, continue with
                           #  next token)
    ],
    [                      # (run conditionally)
        check_token_proper_name,
        translate_proper_name,
        next_token         # (stop processing, continue with
                           #  next token)
    ],
    normalize_token,       # (only for "normal" tokens)
    translate_token,
]
```

Those functions will be passed two parameters (see listing 49): The object, which is currently running through the task sequence, and the engine instance itself.

Listing 49: Interface necessary for a Workflow function [40].

```
def next_token(obj, eng):
    eng.ContinueNextToken()
```

The object parameter is an 'object-relational-mapper'[20](ORM) model instance, which wraps the processed object and adds some additional functionality which is relevant for the persistence. The persistence is currently handled by SQLAlchemy[21] and enforced, but this is an implementation detail likely to be changed in the future. Those ORM model instances basically know how to store themselves as well as their position in the task sequence to return to their execution (It is also possible to restart the execution in a different process).

The engine parameter is also a ORM model instance, but it manages the execution of the Workflow itself. It takes the to be processed objects and passes them to the function in the task sequence.

The task sequence, except for being a sequence of functions that gets executed in order, allows certain control flow mechanics. For example if-else statements as well as loops,

[20] see [41] for more information about object relational mapping

[21] a "Python SQL toolkit and Object Relational Mapper"[42]

even though they are done by jumping to certain points in the task-sequence. Due to those control flow possibilities, the task sequence might actually be considered more of a tree-like structure.

6.1.2 Integration

For showing one successful integration of the automated merging tool as a blueprint for future installments, an Inspire Workflow is chosen that harvests records via the 'Open Archives Initiative'[22] (OAI) protocol from arXiv.org.

The complete Workflow has the form shown in listing 50:

Listing 50: Integration of the automated merging tool in a Workflow.

```
workflow = [
    convert_record_with_repository("oaiarXiv2inspire <...>"),
    convert_record_to_bibfield(model=["hep"]),
    workflow_if(match_record_arxiv_remote_oaiharvest),
    [
        log_info("Record already into database"),
        # automated merging tool part
        workflow_if(update_existing_record_oaiharvest()),
        [
            update_successful_merge
        ],
        workflow_else,
        [
            start_user_interaction,
            finish_user_interaction
        ],
        upload_update,
    ],
    workflow_else,
    [
        workflow_if(match_record_HP_oaiharvest({{ key }})),
        [
            log_info("Record already into Holding Pen"),
            save_identifiers_oaiharvest("HP_IDENTIFIERS"),
            update_old_object("HP_IDENTIFIERS"),
            delete_self_and_stop_processing,
        ],
        workflow_else,
```

[22] see [43] for further information

```
        [
            save_identifiers_oaiharvest("HP_IDENTIFIERS"),
            plot_extract(["latex"]),
            arxiv_fulltext_download,
            classify_paper_with_oaiharvester(
                taxonomy="HEPont",
                output_mode="dict",
            ),
            refextract,
            author_list,
            halt_record_with_action(action="core_approval",
                                    message="Accept article?"),
            workflow_if(was_approved),
            [
                add_core_oaiharvest,
                send_robotupload_oaiharvest(),
            ],
            workflow_else,
            [
                log_info("Record rejected"),
                delete_self_and_stop_processing,
            ],
        ],
    ],
]
```

The relevant part of this Workflow starts in line 8. The function 'update_existing_record_oaiharvester' (see figure 20) fetches the corresponding record and transforms its meta data into the internal data structure. At this point the latest common ancestor is just assumed to be an empty dictionary. The LCA, current production data, the new data and the resolving actions are then passed as arguments to the Merger class. Calling the Merger classes 'run' function posses two possible outcomes: The merge is successful, in which case the Merger object is stored in the 'extra_data' property of the Workflow object and the function returns True. Otherwise, if the 'run' method raises a 'UnresolvedConflictsException', the merger object is also stored in the 'extra_data' property but the function returns False.

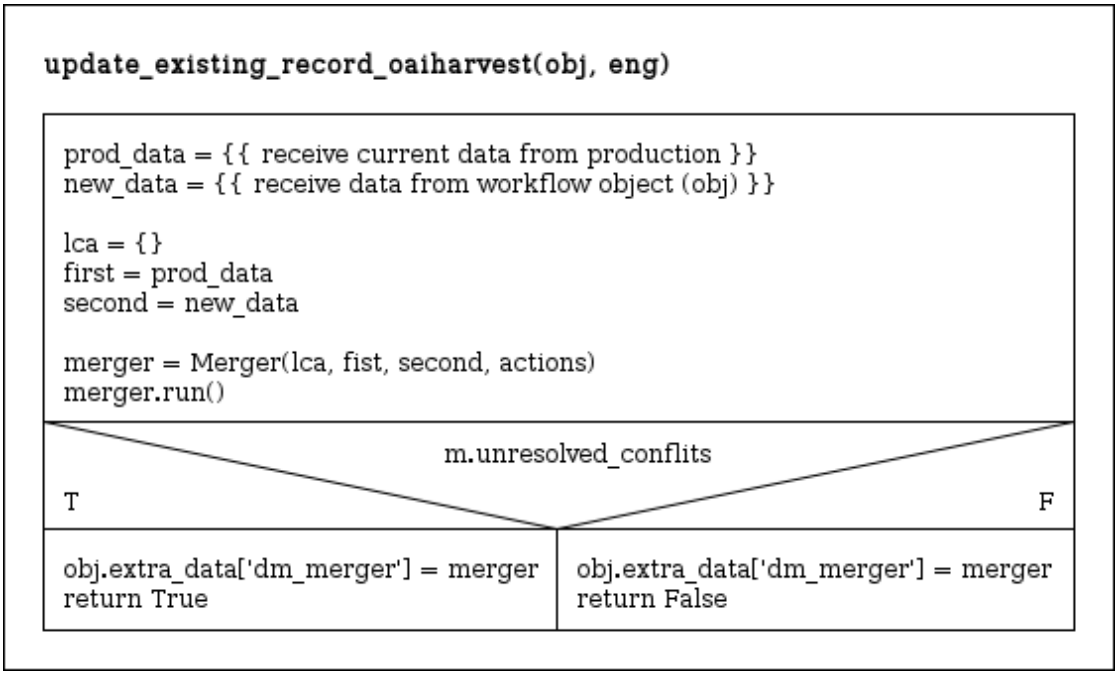

Figure 20: Structure chart showing the steps executed by the 'update_existing_record_oaiharvest Workflow function.

Since the function is called in a 'workflow_if' function, the previous return value determines the ongoing events.

In case of an successful merge, the 'update_successful_merge' function is called, which receives the Merger object from the Workflow object 'extra_data' property and then patches the current production meta data with the unified patches, as shown in figure 21.

update_successful_merge(obj, eng)

```
merger = obj.extra_data['dm_merger']
obj.extra_data['dm_result'] = patch(merger.unified_patches, {})
```

Figure 21: Structure chart showing the steps executed by the 'update_successful_merge' Workflow function.

In the case of unresolved conflicts a Holdingpen interaction needs to be prepared. Therefore, the 'start_user_interaction' function gets called (figure 22), which wraps the content of the 'unresolved_conflicts' property of the merger object in a dictionary, which makes it accessible for the Holdingpen interface (mainly the Jinja templating engine). After this preparation step, the whole Workflow process is halted, and the corresponding Holdingpen action is assigned.

start_user_interaction(obj, eng)

```
merger = obj.extra_data['dm_merger']

# a utility function that maps the unresolved conflict in a way
# manageable by Jinja
res = build_conflict_display(merger.unresolved_conflicts)

obj.extra_data['dictdiffer_unresolved_conflicts'] = res
eng.halt('Resolve please', action='dictdiffer_unresolved_approval')
```

Figure 22: The 'start_user_interaction' Workflow function.

The completion of the Holdingpen action will continue the Workflow, which then executes the 'finish_user_interaction' function. This function receives the picks which were set by the Holdingpen action, and transforms them to the form necessary for the automated merging tool to work. The function then continues the automated merging process and patches the unified patches to the existing production data.

finish_user_interaction(obj, eng)

```
merger = obj.extra_data['dm_merger']
picks = obj.extra_data['dm_picks']

# Considers the limited possiblities of the current Holdingpen
# action and transfrom the pick to the necessary format
picks = transform_picks(picks, merger)

merger.continue_run(picks)
obj.extra_data['dm_result'] = patch(merger.unified_patches, {})
```

Figure 23: The 'finish_user_interaction' Workflow function.

Simply overwriting the 'data' property of the Workflow object with the new bibliographic meta data will result in an upload to the repository.

6.2 Holdingpen

The Holdingpen interface represents an interface for user interaction associated with Workflow tasks.

Whenever a Workflow reaches a state where human interaction is required, a Holdingpen action is assigned to the Workflow object and the Workflow is halted.

In this state, the Workflow and the processed object is visible in a Holdingpen table, shown in figure 24, which provides some general information as well as the possibility to access a details page. Upon calling this details page, Holdingpen checks for assigned

actions, which are realized as a Python backend script. This backend script executes and renders a part of the details page, which should provide every require interaction to solve the problem at hand.

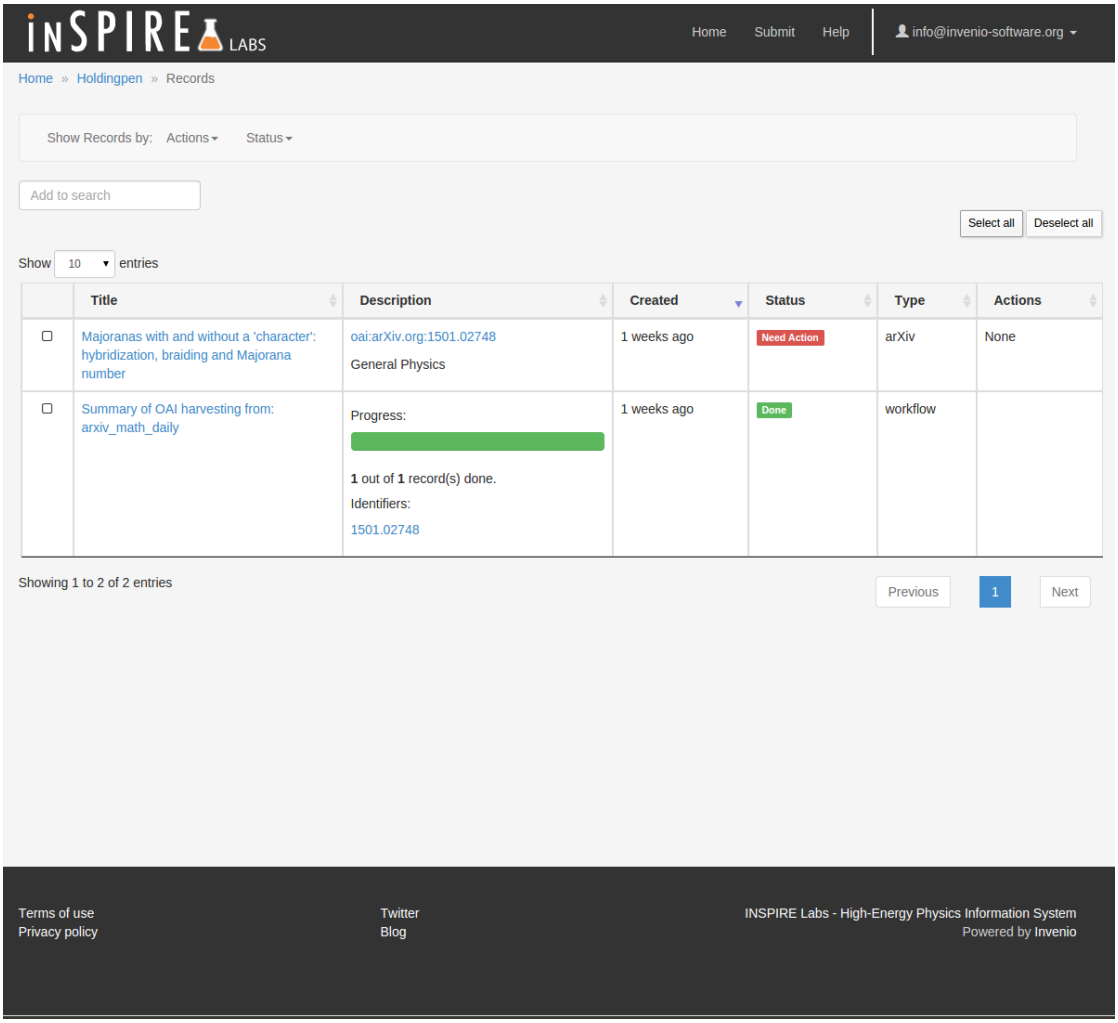

Figure 24: Holdingpen table displaying Workflow actions, that require human interaction.

6.2.1 Details and Usage

Utilizing the Holdingpen is a rather simple process, that doesn't require the user to know too much about the underlying system.

Upon calling the details page of a given Workflow on hold, the Holdingpen module will figure out the assigned Holdingpen action and load the corresponding Python file. This Python file needs to agree to the interface of listing 51:

Listing 51: Holdingpen backend file interface.

```
class {{ CLASS_NAME }}(object):
    def render(self, obj):
        return {"side": {{ HTML as string }},
                "main": {{ HTML as string }}
               }
```

Except for the needed render method, everything is optional and depends on the problem at hand, but there are a few best practice assumptions.

The render function needs to return HTML code as a Python string. How this string is generated is not defined. In Invenio v2, it is common practice to define a Jinja template. Jinja is a template engine that provides certain control flow mechanics like loops and if-else statements to easily generate content [44].

It is also likely that some user interaction is required, which would lead to using Javascript. Yet again, there is no definition of how Javascript should be provided, but invenio pu uses flight.js and require.js to manage the included Javascript code.

Furthermore, since the executed changes need to be brought back to the Invenio instance and into persistence, the Python backend file will need some callback method.

6.2.2 Integration

The Holdingpen integration will be described in two different parts, analogous to the files that need to be set up to make it work. This section will avoid describing the HTML files though.

dictdiffer_unresolved_approval.py
The backend file for the unresolved conflicts handling. As required by the Holdingpen environment, the class implements the necessary method 'render'.

This method returns a dictionary, containing two string representations of HTML markup. This code is created using the Jinja template functionality.

The one string represents the component in the sidebar of the Holdingpen detail view, while the other represents the main content. The sidebar provides the functionality to finish the Holdingpen action, while the main content uses radio buttons to let the user decide which patch to use. See figure 25 for the result of the current implementation.

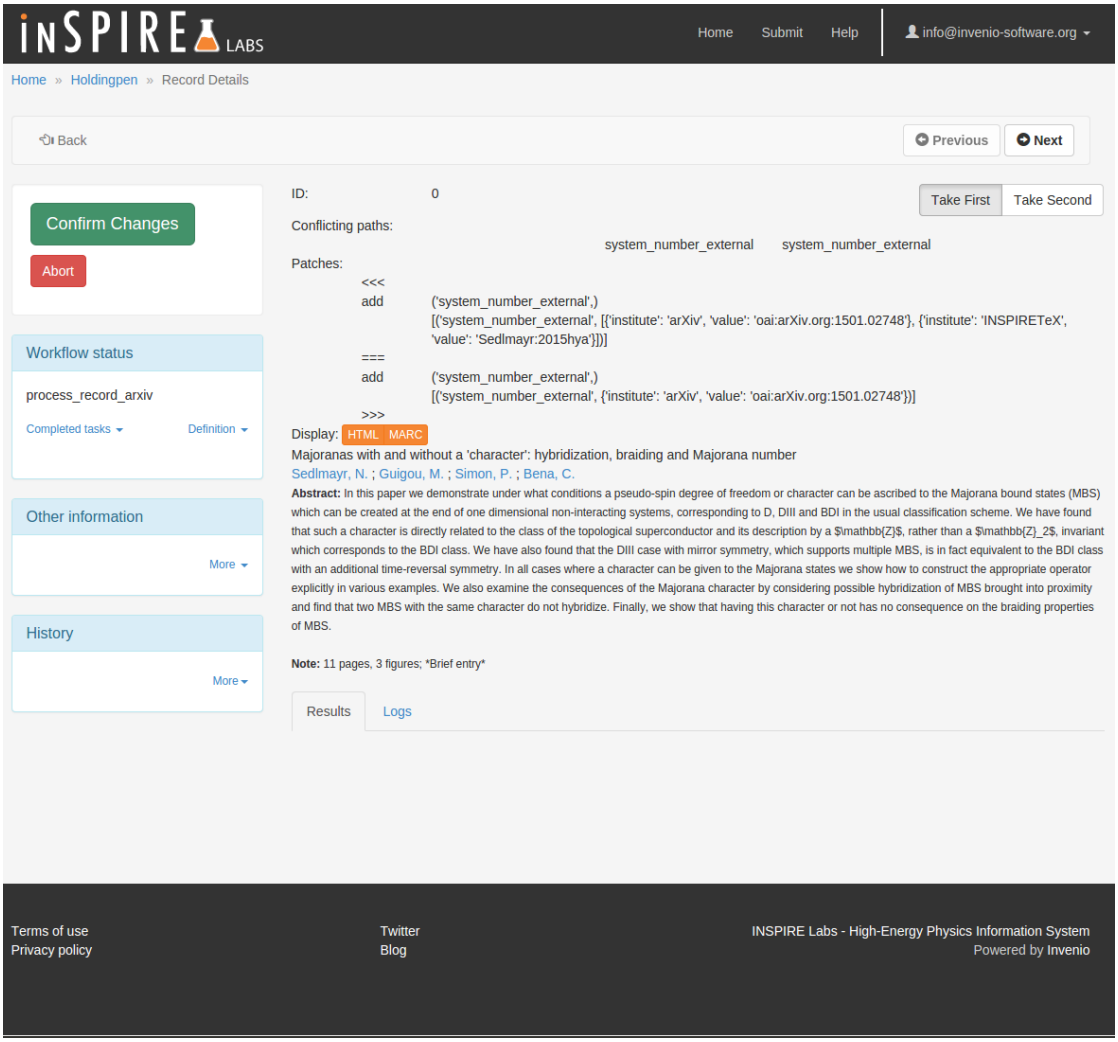

Figure 25: Holdingpen conflict resolution web page providing an interface for human interaction with the merging process.

Additionally, the backend file implements the resolve method, which gets executed upon finishing the Holdingpen action. It extracts the picks made by the user, which are transmitted as a flask request object to the method and stores them in the 'extra_data' property of the Workflow object. It then continues the Workflow and gives the user some visual feedback about the success of the process.

dictdiffer_unresolved_approval.js
This file implements the functionality necessary for a human to select the desired resolution of a given conflict.

This selection process is realized using radio buttons, which allow it in the current implementation to just choose one complete group of patches, either from the first or the second source, in any given conflict.

Upon clicking the 'Confirm Changes' button of the side bar, the confirmation function gets called, that iterates over all active radio buttons, which contain the needed information to build the needed picks.

7 Resolver

In the connection to Invenio based records, most resolvers for certain fields should follow pretty simple rules.

Starting from the currently known examples and use cases, most actions will base their decision on the fact which update is newer, if the source is trusted or if it would overwrite some manually generated content.

But there are at least two notable exceptions: the authors and the references field. The authors and the references fields are fields containing a list of authors or references. Those fields are often affected by amendments. Possible examples would be:

- Additions
- Normalization of names
- Change of format, removal of placeholders
- Addition of meta data
- Change of order
- Change of unique identifiers like IDs

Since most of those changes can be applied by internal or external sources, it is in general not advisable to just take the latest updates coming from an outside source. Additionally, all those changes can make it difficult to pair the corresponding entities. For example: A reordered list of authors with possible additions will be hard to be matched against an old list of authors with normalized names and additional meta data.

Since 'solving' those topics would be enough for a thesis on its own, this section is going to discuss a first implementation, and possible future improvements.

7.1 Assignment Problem

The problem of assigning the entities of two lists is known as the 'assignment problem', which consists of solving the task of calculating a minimum weight matching in a bipartite graph with weighted edges. The assignment problem is a special case of the transportation problem, which is a special case of the min-costflow problem [46].

The 'Hungarian algorithm' is one of many algorithms capable of solving the linear assignment problem in polynomial time.

It should be mentioned that using the Hungarian algorithm might seem like an overkill, but there are cases in the High Energy Physics environment, where a single paper has over 3000 authors. Iterating over those in a simpler approach, worst case being trying every possible combination (which would result in a complexity of O(n!) due to n! permutations [45]), is not going to scale.

7.1.1 Hungarian Algorithm

There are different possible ways to implement and interpret the Hungarian algorithm, which is also called the Munkres-Kuhn algorithm. One way is to see it as a graph problem, the other way is to interpret it as a matrix. This section is going to introduce both ways.

The matrix interpretation
The Hungarian algorithm executed on a cost matrix would require this matrix to be square and operates the following steps (see [45]):

1. Subtract the minimum value in each row from every other entry in that row. This will introduce at least one zero value in those rows.
2. Subtract the minimum value in each column from every other entry in that column. This will introduce at least one zero value in those columns.
3. Every occurred zero value needs to be covered by marking either it's row or column. This coverage needs to be achieved with as few marked rows and columns as possible.
4. If the number of marked rows and columns equals to N, the algorithm terminates and the assignment with the minimal cost was found. Otherwise, continue with the next step.
5. Find the smallest value that is not covered by any marked row or column and subtract it from every unmarked row and add it to every marked column. Continue with step 3.

The graph interpretation
To make the understanding of the following algorithm easier, we will introduce the name 'worker' for every vertex on the 'left' side of the bipartite graph and 'job' for every vertex on the 'right' side of the graph.

The Hungarian algorithm executed on a graph requires it to be bipartite and executes the following steps (see [46]):

1. For each worker, find it's outgoing minimum edge and subtract it's value from every other outgoing edge of that worker. Do the same for each job. This will introduce zero value edges.
2. Find the maximum matching. If it is a perfect matching, the algorithm is finished. Otherwise, continue with the next step.
3. Find the minimum vertex cover 'MVC' using only the zero edges.
4. Adjust the weights using the following formulas 1 and 2 and repeat from step 2.

$$\triangle = min_{i \notin MVC, j \notin MVC}(c_{i,j}) \tag{1}$$

$$c_{i,j} = \begin{cases} c_{i,j} - \triangle & , i \notin MVC \wedge j \notin MVC \\ c_{i,j} & , i \in MVC \vee j \in MVC \\ c_{i,j} + \triangle & , i \in MVC \wedge j \in MVC \end{cases} \tag{2}$$

7.2 Entity matcher

The EntityMatcher class will be utilized to match two lists of entities for further comparison. It will use the Hungarian algorithm and offer a few hooks to be configurable for the specific needs to match authors and references, even though it will try to be as generic as reasonably possible.

7.2.1 Conception

A crucial part about the Hungarian algorithm in this particular case is the calculation of the weights, meaning the distance between two entities of either authors or references. Using simple string similarity algorithms like the Levenshtein distance might be a good start, but doesn't seem suitable because of all the possible changes applicable to the entities. For example, normalizing an authors name by switching the order of name and surname will result in a pretty bad score.

Additionally, calculating the Levenshtein distance for all possible combinations of authors would take quite some time.

For the first implementation, a multilayered approach is proposed:

- Generate a k-gram (probably bi-gram) vector for every author

- Fill in a cost matrix based on those vectors and unique identifiers
- Chose k most likely matches and investigate on them to improve the score

Using this approach utilizing k-grams, the consequences of reordered names gets reduced.

Unique identifiers, like email addresses or reference IDs in the case of authors and references will immediately result in a cost score of zero in the matrix, which means a perfect match. Unfortunately, those identifiers are, in the case of authors, pretty rare and might change in the case of references when the referenced document moves from a preprint record to a published one.

Normalizing author names and affiliations will increase the score quality since they reduce the likeliness of stumbling upon those case. Normalization would probably require a big database of synonyms. This can, however, be implemented over time.

7.2.2 Realization

The EntityMatcher wraps the previously specified steps in one class, which provides a 'run' method to execute all necessary steps and returns the calculated matching of entities. The class provides three hook-ins for customization:

1. Determine keys to build a string representation of the objects.
2. Determine keys that serve as unique identifiers.
3. A custom function that to improve a given score for the similarity of two entities.

For solving the assignment problem, this implementation utilizes the Munkres module [47], since a custom implementation of the needed functionality poses more risks than a field tested module.

7.3 Evaluation

The most important part about the authors and references resolve action is the mapping of the corresponding entities. The robustness and applicability of the chosen approach using the Hungarian algorithm will need to proof itself in real life scenarios eventually, but the following test might provide some estimation.

There are three different cases of amendments that can happen for either the authors or the references of a bibliographic record:

- Entities are changed

- Entities are added
- Entities are removed

The latter two cases using the current implementation lead to the removed or added entities to not being part of the matching. The test will therefore just consider entities that get altered.

To test the robustness of the approach, existing data of the ATLAS collaboration will be used. A certain subset of those authors will be taken and randomly altered to an adjustable degree. The EntityMatcher then tries to calculate the correct matching.

This process is repeated 100 times for all combinations of the following set of parameters:

- Size of the subset: 10, 50, 100, 250
- Rate of alteration: 0.2, 0.4, 0.6, 0.8
- Excluded keys: [], ['INSPIRE_number']
- ID keys: [], ['INSPIRE_number']

Table 7 shows the resulting success rate.

Size	Alteration Rate	Excluded Keys	ID Keys	Success Rate
10	0.2	''	''	1.0
10	0.2	''	'INSPIRE_number'	1.0
10	0.2	'INSPIRE_number'	''	1.0
10	0.2	'INSPIRE_number'	'INSPIRE_number'	1.0
10	0.4	''	''	1.0
10	0.4	''	'INSPIRE_number'	1.0
10	0.4	'INSPIRE_number'	''	1.0
10	0.4	'INSPIRE_number'	'INSPIRE_number'	1.0
10	0.6	''	''	1.0
10	0.6	''	'INSPIRE_number'	1.0
10	0.6	'INSPIRE_number'	''	1.0
10	0.6	'INSPIRE_number'	'INSPIRE_number'	1.0
10	0.8	''	''	1.0
10	0.8	''	'INSPIRE_number'	1.0
10	0.8	'INSPIRE_number'	''	1.0
10	0.8	'INSPIRE_number'	'INSPIRE_number'	1.0
20	0.2	''	''	1.0
20	0.2	''	'INSPIRE_number'	1.0
20	0.2	'INSPIRE_number'	''	1.0

20	0.2	’INSPIRE_number’	’INSPIRE_number’	1.0
20	0.4	”	”	1.0
20	0.4	”	’INSPIRE_number’	1.0
20	0.4	’INSPIRE_number’	”	1.0
20	0.4	’INSPIRE_number’	’INSPIRE_number’	1.0
20	0.6	”	”	1.0
20	0.6	”	’INSPIRE_number’	1.0
20	0.6	’INSPIRE_number’	”	1.0
20	0.6	’INSPIRE_number’	’INSPIRE_number’	1.0
20	0.8	”	”	1.0
20	0.8	”	’INSPIRE_number’	1.0
20	0.8	’INSPIRE_number’	”	1.0
20	0.8	’INSPIRE_number’	’INSPIRE_number’	1.0
50	0.2	”	”	1.0
50	0.2	”	’INSPIRE_number’	1.0
50	0.2	’INSPIRE_number’	”	1.0
50	0.2	’INSPIRE_number’	’INSPIRE_number’	1.0
50	0.4	”	”	1.0
50	0.4	”	’INSPIRE_number’	1.0
50	0.4	’INSPIRE_number’	”	1.0
50	0.4	’INSPIRE_number’	’INSPIRE_number’	1.0
50	0.6	”	”	1.0
50	0.6	”	’INSPIRE_number’	1.0
50	0.6	’INSPIRE_number’	”	0.99
50	0.6	’INSPIRE_number’	’INSPIRE_number’	1.0
50	0.8	”	”	1.0
50	0.8	”	’INSPIRE_number’	1.0
50	0.8	’INSPIRE_number’	”	1.0
50	0.8	’INSPIRE_number’	’INSPIRE_number’	1.0
100	0.2	”	”	1.0
100	0.2	”	’INSPIRE_number’	1.0
100	0.2	’INSPIRE_number’	”	1.0
100	0.2	’INSPIRE_number’	’INSPIRE_number’	1.0
100	0.4	”	”	1.0
100	0.4	”	’INSPIRE_number’	1.0
100	0.4	’INSPIRE_number’	”	1.0
100	0.4	’INSPIRE_number’	’INSPIRE_number’	1.0
100	0.6	”	”	1.0
100	0.6	”	’INSPIRE_number’	1.0
100	0.6	’INSPIRE_number’	”	1.0

100	0.6	'INSPIRE_number'	'INSPIRE_number'	1.0
100	0.8	''	''	0.98
100	0.8	''	'INSPIRE_number'	0.98
100	0.8	'INSPIRE_number'	''	0.97
100	0.8	'INSPIRE_number'	'INSPIRE_number'	1.0
250	0.2	''	''	1.0
250	0.2	''	'INSPIRE_number'	1.0
250	0.2	'INSPIRE_number'	''	1.0
250	0.2	'INSPIRE_number'	'INSPIRE_number'	1.0
250	0.4	''	''	1.0
250	0.4	''	'INSPIRE_number'	1.0
250	0.4	'INSPIRE_number'	''	1.0
250	0.4	'INSPIRE_number'	'INSPIRE_number'	1.0
250	0.6	''	''	0.98
250	0.6	''	'INSPIRE_number'	0.98
250	0.6	'INSPIRE_number'	''	1.0
250	0.6	'INSPIRE_number'	'INSPIRE_number'	1.0
250	0.8	''	''	0.9
250	0.8	''	'INSPIRE_number'	0.88
250	0.8	'INSPIRE_number'	''	0.95
250	0.8	'INSPIRE_number'	'INSPIRE_number'	1.0

Table 7: Evaluation table showing the correlation between the size of the author lists, their amendment rate, the excluded keys and the keys used as an identifier.

7.4 Possible integration as a patch

Matching the updated authors and references is only part of what is required for the corresponding resolver to be done. After assignment, the differences need to be extracted and validated.

This leads to a resolver function with the structure of figure 26. After the matching is calculated using the EntityMatcher class, the mapping is examined for its level of 'certainty', since very unlikely matches can be considered as harmful.

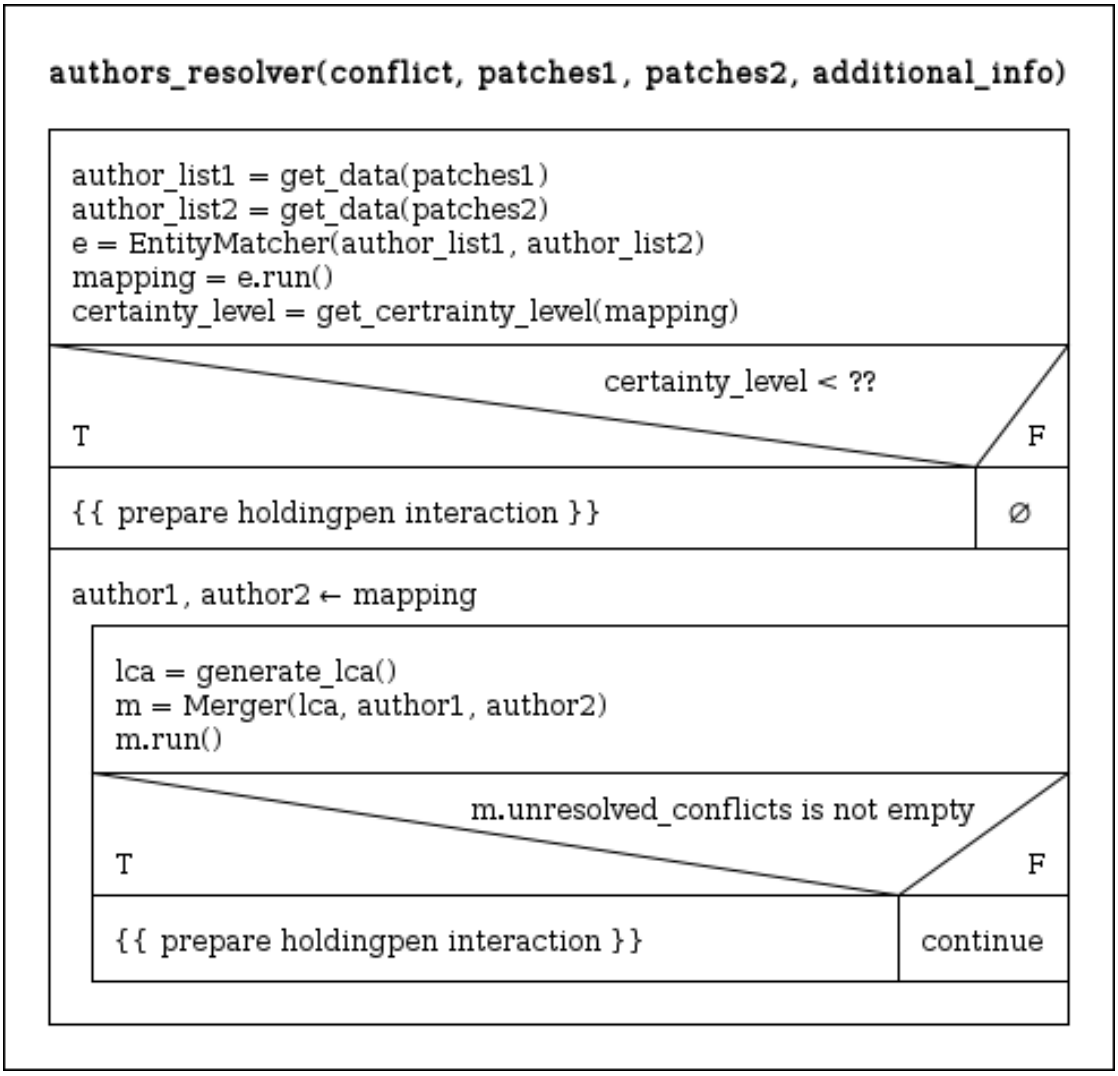

Figure 26: Structure chart showing the implementation of the resolve action handling the authors field.

This basically is a nested automated merge. The matched authors are checked for differences, assuming an latest common ancestor being an empty dictionary. In this cases, the actions required for an automated merge are passed using the 'additional_information' parameter.

It should be noted here, that this approach of merging author lists doesn't utilize the 'take' functionality a lot. Since there are exactly two patches, one containing the authors

list from the first, the other from the second data structure, the 'take' parameter of the Conflict object can just decide on taking one list or the other as a whole. This leads to the necessity of amending the list in place.

Which, in turn, means that this use case is way to complex for the current functionality provided by the Holdingpen action.

In case of a failed merge, the calculated matching is set to the Conflict objects 'manual_resolve' parameter.

8 Additional Usecase - Command Line Interface

This section deal with the additional possible use cases for of an 'command line interface' (CLI) for the automated record merging or parts of it.

Linux' diff command provides a way to see line based difference between two files. While the display of those changes is suitable for most file types, it might not work exceptionally well for XML or JSON files.

This is where a command line tool based on the dictdiffer functionality might come in handy. A path based representation of the differences between two files together with an 'easy' pick/alter patch function could improve the workflow of reviewing those types of files.

To be really useful, this command line tool would need to be able to fix or at least understand merging conflicts of common tools like git.

8.1 Possible use cases

Given the provided functionality by invenio.dictdiffer and the now integrated merging tool, two use cases are quite apparent: Displaying differences between two files and merging conflicting changes.

Those use cases can be divided again into the following sup use cases:

1. Displaying/merging differences between two files.
2. Displaying/merging differences between a conflicted file from version control systems like git.

8.2 Conception

Starting from those two general use cases, the command line interface will be divided into four parts:

1. The argument parser and file loader.
2. The class responsible for displaying the differences.
3. The class responsible for performing the merge.
4. Write the results to a file.

The argument parsing part would simply parse the given arguments and call the corresponding function which handles the rest. This can be easily achieved using Python argparse[23] module. Additionally, it would require an interface to add converter, which would allow to read certain file formats and convert them into a usable Python data structure. Displaying differences and the needed merge operations can then be handled in a generic way.

The DifferenceDisplayer class, which handles the presentation of the differences will operate following this schema:

- Extract the patches.
- Wrap the patches. (Using the SequenceWrapper as default)
- Build a new data structure by traversing the existing one and wrapping eachelement, that is part of a change in an Encoder object. Those encoder object provide a special '__repr__' method, which will then handle the print process.
- print the newly generated data structure.

The MergeDisplayer class would follow this schema:

- Run the merger.
- In case of a unresolved conflicts, follow this loop:
  - Present a brief overview of the conflicting patches.
  - Allow highlighted view of the conflict if necessary.
- Return the merged data structure to write it back to a file.

8.3 A first prototype

The purpose of the first prototype of the CLI is to provide a show case for the possibilities and use cases for the automated merging tool wrapped inside a command line interface. It therefore focuses on the main task, reading, displaying and saving differences and steps of a merging process by reference to example files. Accordingly, the prototype doesn't support a configuration file to hook in possible file reader or writer.

Other than that, it stays close to the rough conception:

The command line interface is divided into three files, cli.py, cli_diff.py and cli_merge.py. The first handles the given command line arguments and then forwards

[23]"Parser for command-line options, arguments and sub-commands"[48]

the task to one of the other two files accordingly. The command line interface differentiates between two sub-commands: 'diff' and 'merge' and provides additional arguments for each of those see listing 52 for all available arguments of the prototype.

Listing 52: List of available arguments for the CLI prototype

```
diff:
    positional arguments:
        first
        second

    optional arguments:
        --type
        --origin

merge:
    positional arguments:
        first
        second
        lca

    optional arguments:
        --type
        --origin
        --target
```

In case of the 'diff' function, the positional arguments 'first' and 'second' are simply the files that are going to be compared. The 'second' argument is not necessary though. In case that it is missing, the 'diff' subcommand will assume that it has to handle a file that contains a git like difference representation. The '–type' argument describes the data structure that is expected to be found in those files. The prototype supports direct JSON, XML and default text. The '–origin' argument can be used if the 'diff' sub command handles a single file, and describes with kind of representation is expected.

The 'merge' function operate similarly. The 'second' and 'lca' arguments are optional, leading to the following behavior:

- If the latest common ancestor file is missing, it is assumed to be an empty data structure of the same kind that is found in first and second.
- If the lca and the second file are missing, it is assumed that the first file contains some kind of difference representation and the lca is assumed to be an empty data structure of the same kind as found in the first file.

The optional argument '–target' directs to a file to store the resulting merge. In case that

it is not given, the result will simply be printed.

The 'diff' sub function itself operates the steps shown in 27:

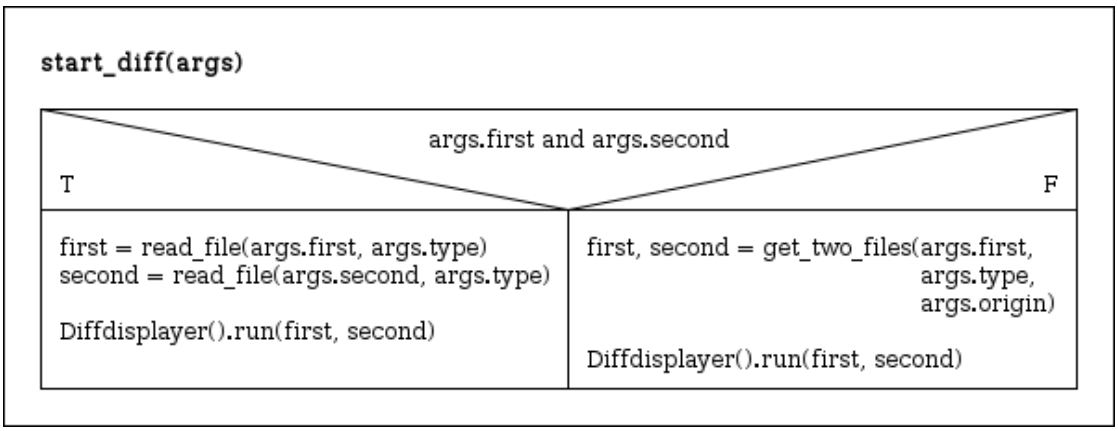

Figure 27: Structure chart of the 'diff' sub function.

The 'DifferenceDisplayer' class is a utility that manages the display of the difference in the data structure. It will therefore use the dictdiffer.diff function to extract the differences between the two given data structures 'first' and 'second' and then applies those patches. But instead of making the direct amendments (as dictdiffer.patch would), it will insert a special 'Change' class object at the patches path. This class utilizes Pythons '__repr__' method, which gets called whenever an object is going to be printed. It will therefore display the differences when the complete data structure get printed. See listing 53 for an example.

Listing 53: Command line interface diff sub function example.

```
First:
    {"changeme": "Hallo",
     "planets": ["Merkur",
                 "Venus",
                 "Erde",
                 "Mars",
                 "Jupitur",
                 "Saturn",
                 "Uranus",
                 "Neptun"],
     "dict": {"a": "b"}}
```

```
Second:
    {"changeme": "Hello",
     "planets": ["Merkur",
                 "Venus",
                 "Earth",
                 "Bob",
                 "Mars",
                 "Saturn",
                 "Neptun",
                 "foo",
                 "bar"],
     "dict": {"a": "x"}}

python cli.py first second --type=json:
    {"changeme": "Hallo -> Hello",
     "planets": ["Merkur",
                 "Venus",
                 "Erde -> Earth",
                 "-> Bob",
                 "Mars",
                 "Jupitur ->",
                 "Saturn",
                 "Uranus ->",
                 "Neptun",
                 "-> foo",
                 "-> bar"],
     "dict": {"a": "b -> x"}
```

The 'merge' sub function itself operates the steps shown in 28:

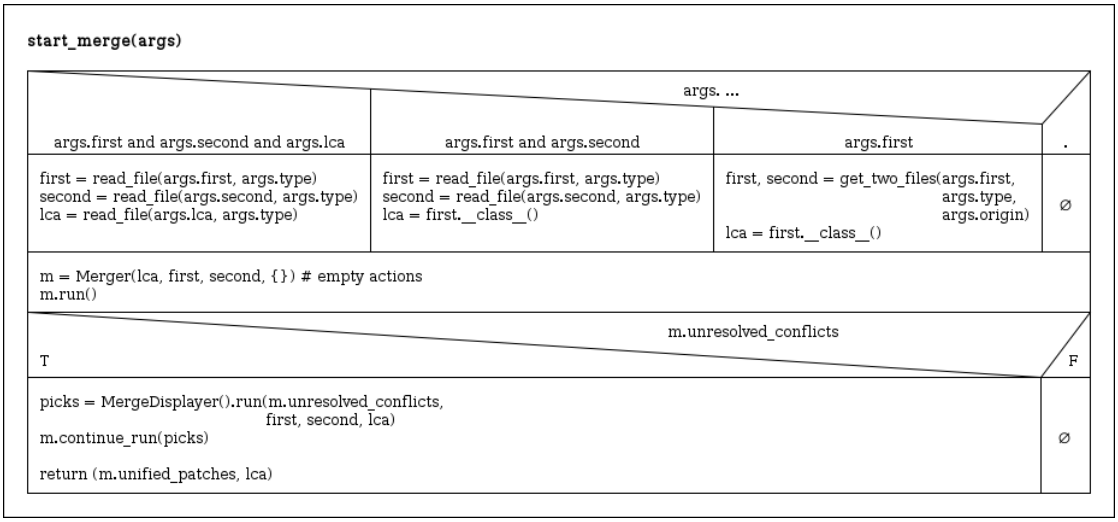

Figure 28: Structure chart of the 'merge' sub function.

The 'MergeDisplayer' classes manages the display and the manual resolution of the unresolved conflicts happening during the process. It does so in a two stage process. At first, it will simply display the conflicting patches in the dictdiffer format, as shown in listing 54 (considering the same files first and second as in 53).

Listing 54: First stage of the command line interface merging process.

```
0
    Pick: NOT PICKED YET
    ('changeme',) ------ ('changeme',)
    <<<<<<<<
    ('add', '', [('changeme', 'Hallo')])
    ========
    ('add', '', [('changeme', 'Hello')])
    >>>>>>>>

1
    Pick: NOT PICKED YET
    ('dict',) ------ ('dict',)
    <<<<<<<<
    ('add', '', [('dict', {'a': 'b'})])
    ========
    ('add', '', [('dict', {'a': 'x'})])
    >>>>>>>>
```

```
2
    Pick: NOT PICKED YET
    <<<<<<<<<
    ('planets',) ------ ('planets',)
    ('add', '', [('planets', ['Merkur',
                               'Venus',
                               'Erde',
                               'Mars',
                               'Jupiter',
                               'Saturn',
                               'Uranus',
                               'Neptun'])])
    ========
    ('add', '', [('planets', ['Merkur',
                               'Venus',
                               'Earth',
                               'Bob',
                               'Mars',
                               'Saturn',
                               'Neptun',
                               'foo',
                               'bar'])])
    >>>>>>>>>
```

From this interface (listing 54), the user can issue the following commands to manually resolve the conflicts:

- 'show id ': This will utilize the DifferenceDisplayer class (described above) to show the differences in the data structures side by side, highlighting only the changes corresponding to the patch.
- 'pick id side : This decides which of the conflicting patches to pick.
- 's' to save the result.
- Utility operations like 'refresh the screen' ('r') and 'quit' ('q').

8.4 More?

It will show with future use and growing experience if the automated merging tool will be suitable for further use cases.

9 Further Development

In the course of this thesis, a working automated merging tool was delivered which proofed itself in certain scenarios, but needs to be evaluated in a real life environment. Regardless of the approaching demands for changes, which will emerge with the real life use, there are certain things already apparent at the current state.

9.1 diff with speed

Certain records that will be handled by the automated merging tool reach sizes up to 700KB. Handling those files requires time, so investing into speeding up the differences extraction process might pay off at a certain point.

One possible attempt would be to transform the dictdiffer.diff function from a recursive to an iterative function. This would eliminate the overhead of additional function calls. The actual speed gain remains questionable for the case of Invenio though, since the bibliographic meta data originates in the MARCXML format, which is more of a list than a deeply nested structure, which might keep the number of recursions already low. This would have to be evaluated using real life data.

Another attempt would Cython, which allows writing C extensions for Python[49]. It utilizes the Cython compiler, to translate Python code into C code, thereby producing highly optimized and compiled code.

9.2 merge with speed

Similar to speeding up the extraction process, speeding up the merging might also be beneficial.

Except for using Cython as in the difference extraction, there is one particular part of the merging process which seems pretty ineffective, and that is the search for conflicts. In the current version of the automated merging tool, it is a comparison of each patch of the first data structure with each patch of the second. This takes O(n*m) to finish.

One improvement could be to build an index, which would be queried for each path to find potential conflicts. Building this conflict would be possible in the complexity of O(n+m) and finding all conflicts would require the same time in the worst case, resulting in O(2*(n+m)).

9.3 Different algorithm for the assignment problem

The Hungarian algorithm has a complexity of O(nˆ4)[46], which can be pretty slow for lists with more than 3000 elements. Finding a robust alternative to the current implementation might result in a big speed gain.

Additionally, creating the cost matrix for the Hungarian algorithm also takes a lot of time, even though it has a complexity of O(n*m). The bottleneck here could be the calculation of the distance between two vectors. Utilizing Numpy[24] or, again, Cython could speed up the process significantly.

9.4 UX/UI

The section that requires the most effort would be the improvement of the UX/UI.

The proof of concept for the holdingpen interface is looking poor and provides functionality that is behind the current possibilities of the automated merging tool. Adding the missing functionality wouldn't be a lot of effort, but doing so in a user friendly way is challenging.

Since the current display of conflicts is pretty close to the technical implementation, talking about paths and not even stripping Python's list indicators a proper user interface concept is needed, hopefully backed up with preceding user testing.

The same is true for the command line interface, even though it is probably not that urgent in this case. The audience for those kinds of tools is likely to be close to some kind of programming anyway, so a somewhat steeper learning curve might be excused. Nonetheless, more work needs to be put into this field.

[24] a "package for scientific computing with Python"[50]

10 Conclusion

In the course of this thesis and automated merging tool was realized based upon existing functionality in the form of the invenio.dictdiffer module from the environment, where the functionality is eventually needed.

Considering the available material, the created module should be able to fulfill its task of merging bibliographic meta data, even though future, real life test cases will ultimately show if it provides all the necessary functionality or not.

Additionally, this thesis realized proof of concept integrations using the example of inspirehep and therefore provided insights of possible future approaches to use the module. It also showed one additional use case in form of a command line interface, which sadly didn't manage to leave an early conceptual state and is therefore not part of the module as of this writing.

Further improvements that are listed in the 'Further Development' section are hopefully going to be considered as soon as the module proofed its worth with some real life scenarios.

References

[1] Oxford Dictionaries, Curator, Retrieved: 18.02.2015, From: http://www.oxforddictionaries.com/definition/english/curator?searchDictCode=all

[2] CERN, About CERN, Retrieved: 17.02.2015, From: http://home.web.cern.ch/about

[3] CERN, The Large Hadron Collider, Retrieved: 17.02.2015, From: http://home.web.cern.ch/topics/large-hadron-collider

[4] CERN, ATLAS, Retrieved: 17.02.2015, From: http://home.web.cern.ch/about/experiments/atlas

[5] CERN, CMS, Retrieved: 17.02.2015, From: http://home.web.cern.ch/about/experiments/cms

[6] CERN, CERN publications in 2013, Retrieved: 20.02.2015, From: http://library.web.cern.ch/annual/cern-publications-2013

[7] SCOAP3.org, Welcome Retrieved: 20.02.2015, From: http://scoap3.org/

[8] Clement Romeu, Anne Gentil-Beccot, Anne Mansuy, Salvatore Mele, Martin Vesper, The SCOAP3 initiative and the Open Access Article-Processing-Charge market: global partnernship and competition improve value in the dissemination of science, 04.06.2014

[9] INSPIRE About Inspire, Retrieved: 17.02.2015, From: https://inspirehep.net/info/general/project/index

[10] INSPIRE Inspire citation metric, Retrieved: 17.02.2015, From: https://inspirehep.net/help/citation-metrics?ln=en

[11] arXiv, About arXiv, Retrieved: 17.02.2015, From: http://arxiv.org/help/general

[12] arXiv, arXiv submission rate, Retrieved: 17.02.2015, From: http://arxiv.org/stats/monthly_submissions

[13] Invenio, Abotu Invenio, Retrieved: 17.02.2015, From: http://invenio-software.org/

[14] NISO, (2004), Understanding Metadata, NISO Press

[15] Matt Schudel, Henriette Avram, 'Mother of MARC', Dies, Retrieved: 17.02.2015, From: http://www.loc.gov/loc/lcib/0605/avram.html

[16] LOC, Standards at the Library of Congress, Retrieved: 17.02.2015, From: http://www.loc.gov/standards/

[17] Wikipedia, MARC Standards, Retrieved: 17.02.2015, From: http://en.wikipedia.org/wiki/MARC_standards

[18] Library of Congress, A Summary of Commonly Used MARC 21 Fields, Retrieved: 17.02.2015, From: http://www.loc.gov/marc/umb/um07to10.html

[19] Jiri Kuncar, Lars Holm Nielsen, Tibor Simko, Open Repositories 2014, Helsinki, Finland, Invenio v2.0: A Pythonic Framework for Large-Scale Digital Libraries,

[20] Introducing JSON, Retrieved: 17.02.2015, From: http://www.json.org/

[21] Wikipedia, Elasticsearch, Retrieved: 18.02.2015, From: http://en.wikipedia.org/wiki/Elasticsearch

[22] mongoDB, Retrieved: 18.02.2015, From: http://www.mongodb.org/about/introduction/

[23] git, Getting Started - About Version Control Retrieved: 18.02.2015, From: http://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

[24] Wikipedia, Git (Software), Retrieved: 18.02.2015, From: http://en.wikipedia.org/wiki/Git_%28software%29

[25] Oxford Dictionaries, Retrieved: 17.02.2015, From: http://www.oxforddictionaries.com/definition/english/merge

[26] Wikipedia, Merge (revision_control), Retrieved: 17.02.2015, From: http://en.wikipedia.org/wiki/Merge_(revision_control)

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to algorithms (3rd edition), 2009, The MIT Press

[28] UCI School of Information and Computer Science Design and Analysis of Algorithms Lecture notes for February 29, 1996 Retrieved: 17.02.2015 From: http://www.ics.uci.edu/ eppstein/161/960229.html

[29] Brahm Cohen, The criss_cross merge case Retrieved: 17.02.2015 From: http://www.gelato.unsw.edu.au/archives/git/0504/2279.html

[30] Pypi, dictdiffer 0.3.0, Retrieved: 17.02.2015, From: https://pypi.python.org/pypi/dictdiffer/

[31] Python.org, Functional Programming HOWTO, Retrieved: 18.02.2015, From: https://docs.python.org/2/howto/functional.html

[32] Python.org, Generators, Simplified Code, Retrieved: 17.02.2015, From: https://wiki.python.org/moin/Generators

[33] Eric Lippert, What is ”duck typing”? Retrieved: 19.02.2015, From: http://ericlippert.com/2014/01/02/what-is-duck-typing/

[34] Python.org, difflib - Helpers for computing deltas, Retrieved: 17.02.2015, From: https://docs.python.org/2/library/difflib.html

[35] John W. Ratcliff, David E. Metzener, PATTERN MATCHING: THE GESTALT APPROACH, Retrieved: 17.02.2015, From: http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1988/8807/8807c/8807

[36] Mads Soegaard, Gestalt principles in form perception, Retrieved: 19.02.2015, From: https://www.interaction-design.org/encyclopedia/gestalt_principles_of_form_perception.html

[37] Sanjeev Khanna, Keshav Kunal, Benjamin C. Pierce, A Formal Investigation of Diff3, Retrieved: 19.02.2015, From: http://www.cis.upenn.edu/ bcpierce/papers/diff3-short.pdf

[38] Martin Fowler, Rebecca Parsons, Domain Specific Languages, Retrieved: 19.02.2015, From: http://martinfowler.com/books/dsl.html

[39] Python.org, ConfigParser - Configuration file parser, Retrieved: 19.02.2015, From: https://docs.python.org/2/library/configparser.html

[40] workflow.readthedocs.org, Worklfow, Retrieved: 17.02.2015, From: http://workflow.readthedocs.org/en/latest/

[41] Hibernate.org, What is Object/Relational Mapping? Retrieved: 19.02.2015, From: http://hibernate.org/orm/what-is-an-orm/

[42] SQLAlchemy.org, Retrieved: 19.02.2015, From: http://www.sqlalchemy.org/

[43] Open Archive Initiative, Retrieved: 21.02.2015, From: http://www.openarchives.org/

[44] Jinja, Retrieved: 20.02.2015, From: http://jinja.pocoo.org/

[45] Harvard, The Assignment Problem and the Hungarian Method, Retrieved: 17.02.2015, From: http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf

[46] topcoder, Assignment Problem and Hungarian Algorithm, Retrieved: 17.02.2015, From: http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=hungarianAlgorithm

[47] Pypi, Munkres 1.0.7, Retrieved: 17.02.2015, From: https://pypi.python.org/pypi/munkres/

[48] Python.org, argparse - Parser for command-line options, arguments and sub-commands, Retrieved: 20.02.2015, From: https://docs.python.org/3/library/argparse.html

[49] cython.org, Cython C-Extensions for Python, Retrieved: 20.02.2015, From: http://cython.org/

[50] NumPy.org, NumPy, Retrieved: 20.02.2015, From: http://www.numpy.org/

Listings

List of Figures

List of Tables