Mapping by Example: Towards an RML Mapping Reverse Engineering Pipeline

Michael Freund[1,*], Rene Dorsch[1], Sebastian Schmid[2] and Andreas Harth[1,2]

[1]Fraunhofer Institute for Integrated Circuits IIS, Nürnberg, Germany
[2]Friedrich-Alexander-Universität Erlangen-Nürnberg, Nürnberg, Germany

Abstract

We introduce a reverse engineering pipeline to generate an RML mapping document from a given non-RDF source and an expected RDF graph. We present and discuss the core algorithms required to implement the reverse engineering pipeline, and demonstrate the algorithms in a prototypical implementation called ReMap. The proposed reverse engineering approach enables users to convert non-RDF data into RDF by example. Users provide an example RDF output graph based on non-RDF input, and the pipeline automatically generates an RML mapping document that transforms the non-RDF input into the desired RDF graph. Additionally, the approach allows updating existing RML mapping documents from older standards to the latest RML vocabulary by using the original non-RDF data and the already mapped RDF data to reverse engineer a corresponding RML mapping document using the latest standard. The ReMap tool is evaluated for conformance to the specification using the RML core test cases and compared to a similar approach using a Large Langauge Model for RML mapping document generation. Additionally we evaluated the performance in terms of execution time and memory consumption using a benchmark dataset. The results show that the ReMap tool conforms to all applicable test cases, while an LLM-based approach performs 31% worse. The performance results show that the ReMap tool exhibits a time complexity of $\mathcal{O}(n \cdot q)$, where n represents the number of non-RDF input elements and q denotes the number of RDF terms in the target RDF graph.

Keywords

RDF Mapping Language (RML), RML Mapping Generation, Knowledge Graph Construction

1. Introduction

The Resource Description Framework (RDF) [1] is the data model used in Knowledge Graphs (KGs) [2]. To integrate structured data (e.g., CSV) or semi-structured data (e.g., JSON) into a KG, users can use the RDF Mapping Language (RML) [3] to transform the non-RDF data to RDF. The RML ecosystem already provides several tools that attempt to simplify RML mapping generation, including the RMLEditor [4], a user-friendly low-code editor for RML mappings, and YARRRML [5], a more human-readable, YAML-based mapping language, able to be translated into RML. Additionally, YARRRML is supported by Matey [6], a dedicated editor that simplifies mapping creation and maintenance.

Despite the availability of tools to assist users, creating RML mappings remains a challenge. Users must consider the available input data and define the desired structure and ontologies of the target RDF graph. Only after that the users can write RML mappings that describe the necessary data transformations to bridge structured or semi-structured input data to the desired RDF graph. The creation of the RML mappings requires familiarity with RDF graph modeling, as well as an understanding of RML ontology terms [7] and syntax.

Introducing a reverse engineering approach to generate RML mappings based on non-RDF source data and the expected output RDF graph can simplify the mapping creation process. The approach eliminates the need to manually write RML mappings after defining the target RDF graph, making the transformation workflow more efficient. The approach allows users to map non-RDF source data *by*

KGCW'25: 6th International Workshop on Knowledge Graph Construction, June 1 or 2, 2025, Portoroz, SVN
*Corresponding author.
michael.freund@iis.fraunhofer.de (M. Freund)
0000-0003-1601-9331 (M. Freund); 0000-0001-6857-7314 (R. Dorsch); 0000-0002-5836-3029 (S. Schmid); 0000-0002-0702-510X (A. Harth)

example, meaning that users provide the input source data and a comprehensive example of the desired target RDF graph, and let the system reverse engineer the required RML mapping document.

The reverse engineering approach can be difficult because different RML mapping documents using different RML language constructs can transform the same non-RDF source data into an equivalent output RDF graph, requiring identification of the most general triple maps. That is, triple maps in the reverse engineered RML mapping document that generate duplicate RDF output triples must be identified and filtered out to generate a minimal and comprehensive RML mapping document.

Previous work has focused on either generating RML mappings by identifying common concepts between non-RDF source data and a given ontology [8], or performing the inverse transformation of converting RDF data to a non-RDF format using its corresponding RML mapping [9]. In contrast, we aim to reverse engineer and generate RML mapping documents directly from non-RDF input and the expected RDF output.

In our approach, we introduce a reverse engineering pipeline that generates a normalized RML mapping document containing RML triple maps that describe the transformation of a given non-RDF input data into a corresponding RDF output. The pipeline first identifies the term types, term map types, and term maps of all RDF terms in the output RDF graph by performing string comparison operations on the input data. Based on the extracted information, a search space of RML triple maps describing the generation of each triple in the output RDF graph is generated. In the second step, the pipeline extends this search space by grouping RML triple maps that describe potential joins. In the third and final step, the pipeline processes the search space and identifies the most general triple maps to generate a minimal mapping document by aggregating RML triple maps that generate identical RDF triples, thus avoiding redundant generation. The final RML triple maps are then written to disk and stored in the resulting RML mapping document. Because the approach is based solely on deterministic operations, such as string comparisons, the generation of RML mappings is transparent.

The key contributions of this work are:

- The introduction of a reverse engineering pipeline to generate RML mapping documents based on non-RDF source data and expected RDF output.
- The introduction of algorithms to derive the term map, the term map type and the term type.
- The empirical conformance and performance evaluation of a proof-of-concept implementation using the RML core test cases and a benchmark dataset, with a comparison to an LLM-based approach.

2. Related Work

Transforming non-RDF data to RDF using RML typically involves two files, the input source data and the RML mapping document, and generates a third file containing the RDF graph. This workflow is implemented by various RML interpreters such as Morph-KGC [10], SDM-RDFizer [11], RMLStreamer [12], or FlexRML [13]. Previous work has explored the inversion of the typical process where the two given files are instead the RML mapping document and a RDF graph and the aim is to generate the non-RDF data as output [9], showing results with limitations. The latest research in converting RDF data to non-RDF formats has instead focused on defining new mapping languages and techniques [14]. In contrast to these approaches we want to focus on the generation of a RML mapping document, with the given files being the non-RDF source data and the RDF output graph.

The automatic generation of RML mappings is only explored by a limited number of publications. Previous work has investigated how RML mapping documents can be generated based on a given non-RDF source data and a target ontology. On the on hand are approaches, trying to match concepts in the ontology to the given non-RDF data [8]. The method is related to approaches used in semantic table annotation, where tabular data is annotated based on information in KGs [15, 16]. On the other hand, since the rise of Large Language Models (LLMs) such as the Gemma family of models [17] or the GPT series [18] with their ability to process natural language text [19], research has been exploring the use of LLMs in RML mapping generation. In [20] a LLM-based pipeline for ontology development

is introduced. The pipeline consists of a mapping component for RML generation to describe the transformation between the developed ontology and the source dataset. In [21], an LLM-based RML mapping document generation pipeline is introduced that uses a target ontology and non-RDF source data as input, generates an RML mapping document in Turtle syntax, and validates and repairs the generated Turtle files. These approaches are either machine learning based or use different similarity metrics to fully automate the generation process, requiring only the target ontology and the non-RDF data. In contrast we want to introduce a deterministic approach that generates reliably RML mapping documents using string comparision operators, so we require the non-RDF data and the target RDF graph as input. Furthermore, our goal is not to fully automate the generation of RML mappings, but rather to provide a pipeline that can help users generate mappings based on a given set of non-RDF source data and a comprehensive example of the target RDF graph.

The concept closest related to our approach is query reverse engineering [22] from the field of relational databases. The aim of query reverse engineering is to generate an SQL query given a database and a result table. The generated SQL query must be instance-equivalent to the original unknown query used to generate the result table. The approach allows for the identification of alternative queries over the data and the identification of unknown data connections. Additionally, query reverse engineering allows users to query the data by example, where users provided examples of the desired data and let the system reverse engineer the query [23]. In our approach, we want to transfer the concepts of query reverse engineering to the domain of KG construction, in order to allow users to map non-RDF data to RDF by example.

3. Preliminaries

Typical RML interpreters transform the set of non-RDF input data, $I = \{i_1, i_2, \ldots, i_n\}$, to RDF output, G, by interpreting the operations described by the set of triple maps $\{t_1, t_2, \ldots, t_n\}$ contained in the RML mapping document M and applying it to I. The interpreter therefore processes multiple input files using triple maps in an RML mapping document to produce an RDF output. The transformation can be formalized as a function f such that $f(M, I) = G$. The function f is implemented by RML interpreters and essentially applies string replacement operations, such as filling in string templates, directly inserting data using references, and performing string formatting. The string formatting operations ensure that the output is a valid IRI (enclosed in `<>`), a valid blank node (starting with `_:`), or a literal (enclosed in `""`) with an appropriate data type or language tag.

Fig. 1 shows the entire mapping process. The non-RDF source data is in CSV format, and the corresponding RML mapping document[1] uses the latest RML vocabulary. Both are used as input to an RML interpreter, which generates the output RDF graph in N-Triple format. The non-RDF source data, the RML mapping document, and the RDF output will be used throughout this paper as a running example.

4. Approach

In our approach we aim to invert the function $f(M, I) = G$ typically implemented by RML interpreters, as introduced in the previous section. Instead of starting with an RML mapping document M to produce the RDF output G from the non-RDF source data I, we reverse the process. Given the set of non-RDF input data I and the RDF output G, our goal is to generate a minimal RML mapping document M' that is equivalent to the original and unknown mapping document M. This means that the RML mapping documents M and M' produce identical RDF output graphs, i.e., $f(M, I) = f(M', I) = G$, while potentially using different RML mapping constructs. M' and M are both possible RML mapping documents within the set of all equivalent RML mapping documents, denoted as V, that is $M', M \in V$. In some cases, M' and M may also represent the same element, i.e., $M' = M$. Our approach aims to identify the $M' \in V$ that contains the minimal number of RML triple maps.

[1]Well-known prefixes are omitted, but can be looked up on http://prefix.cc/.

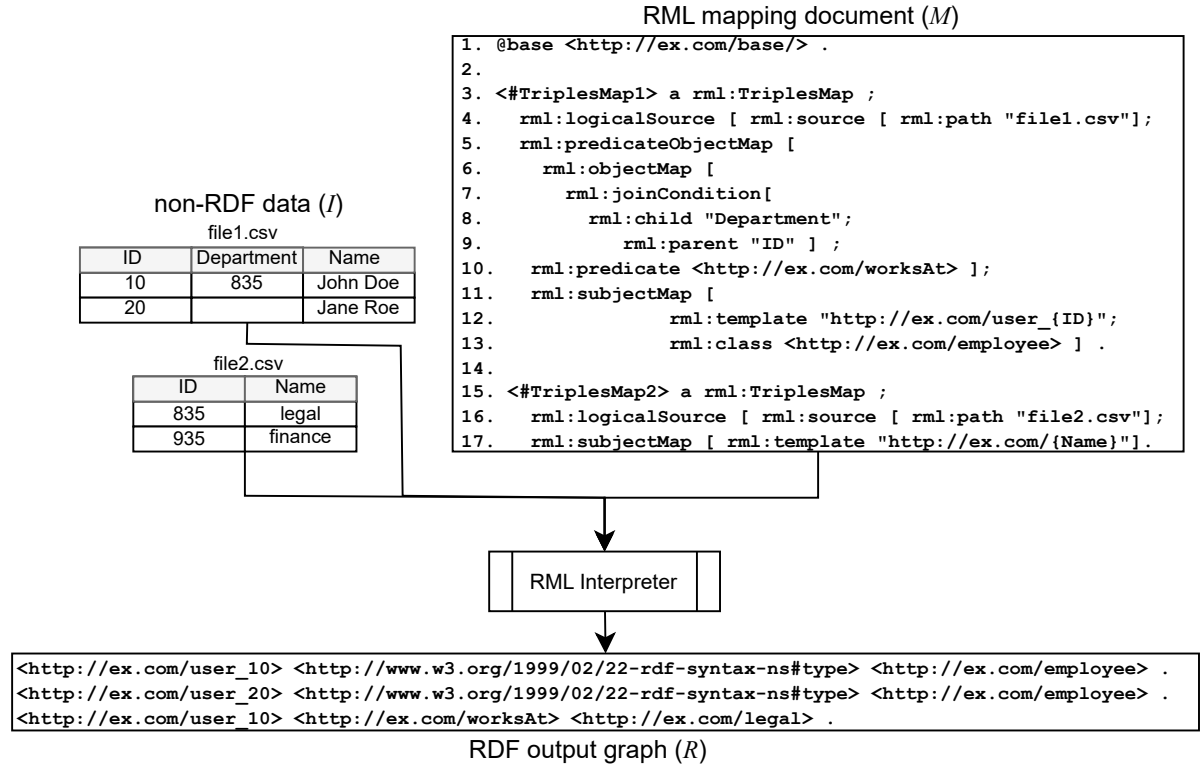

Figure 1: Running example with input data $I = \{file1.csv, file2.csv\}$, RML mapping document M, and output RDF graph G

.

Given the non-RDF input and the RDF output we need to determine the three components `term type`, with possible values of IRI, blank node, and literal, `term map type` with values of template, reference, and constant, and `term map` which consists of strings to be processed based on all output RDF triples. The information is necessary to generate all potential RML triple maps, thereby creating a search space of triple maps. By evaluating this search space, we aim to identify the minimal RML mapping document M' by selecting the best-fitting RML triple maps. Specifically, we prefer triple maps that use a term map type of template or reference over constants, as they typically generate more RDF triples in the output. The identified best-fitting RML triple maps are then included in the final RML mapping document M', ensuring that the final result contains the minimal necessary number of triple maps. The overall reverse engineering pipeline is depicted in Fig. 2. The pipeline generally consists

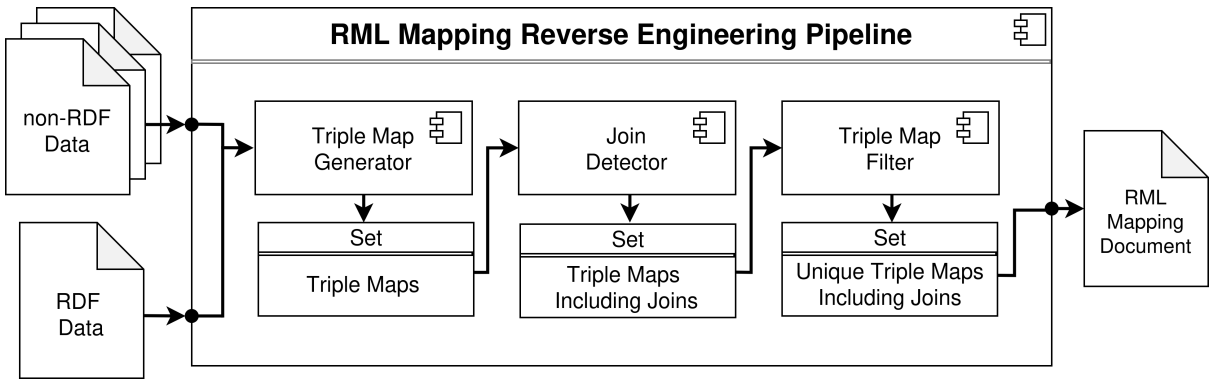

Figure 2: Overview of core components in the RML mapping document reverse engineering pipeline.

of three processing components. The first component, the *Triple Map Generator*, is used to identify

term types, term map types, and term maps. The information is used to generate all possible triple maps, which creates the initial search space, represented as a set of triple maps. The second component, the *Join Detection* component, takes the set of triple maps as input and identifies potential joins by combining related triple maps into a single join triple map. The component therefore extends the initial search space and produces a new set of triple maps that includes the identified joins as output. The final component, the *Triple Map Filter*, processes the created search space and identifies and filters all triple maps that produce a subset of RDF triples from other triple maps, with the goal of identifying all triple maps required to generate a minimal RML mapping document M'. Finally, all valid triple maps are combined into the generated RML mapping document and saved to disk. All three components and their computational complexity are discussed in more detail in the following subsections.

4.1. Triple Map Generator

The Triple Map Generator component takes the set of non-RDF source data I and the target RDF graph G as input. Using the following three algorithms, the component extracts all term types, term map types, and term maps. Based on the extracted information and predefined triple map templates, the component generates a set of potential RML triple maps as output, representing the initial search space.

Identifying Term Types The first step in the Triple Map Generator component of the reverse engineering pipeline is to determine the term types of the subject, predicate, object, and optional graph terms of the output RDF graph. The term types are identified based on the formatting of RDF terms in different RDF serializations. We assume that the serialization format is N-Triple, since this is the format most commonly generated by RML interpreters. In N-Triples, blank nodes are prefixed with `_:`, IRIs are enclosed in `< >`, and literals are enclosed in `" "`. Algorithm 1 describes the identification process in pseudocode and assumes well-formed RDF terms as input. The algorithm is relatively simple but must be executed for each RDF term in the output RDF graph.

Algorithm 1: Term Type Identification

Input: RDF term T (string)
Output: Identified RDF term type (string)

```
// Remove language tag or datatype
if '^^' in T then
    T ← T.split('^^')[0];
else if '@' in T then
    T ← T.split('@')[0];
// Identify term type
if T[0] == '<' and T[-1] == '>' then
    return iri;
else if T[0] == '_' and T[1] == ':' then
    return blanknode;
else if T[0] == '"' and T[-1] == '"' then
    return literal;
else
    error
```

For instance, when Algorithm 1 processes all RDF terms in the output RDF graph of the running example introduced in Section 3, it determines that all RDF terms are of term type `IRI`.

Identifying Term Map Types The next step is the identification of the term map type in the output RDF data. The term map type of an RDF term is determined based on how the data is generated.

Specifically:

- If the RDF term does not contain values from the non-RDF source data, its term map type is `constant`.
- If the RDF term contains only values from the non-RDF source data, its term map type is `reference`.
- If the RDF term contains some values from the non-RDF source data, its term map type is `template`.

To avoid identifying template values within well-known IRIs, we introduce protected IRIs, a predefined list of IRIs that are considered protected, meaning that no replacement operations are performed on them. The list can be customized for a specific domain, and relevant protected IRIs can be added. Additional processing is required if the term type is an IRI, since the algorithm must first decode the IRI, i.e. remove percent encoded characters, before it can perform data matching. Algorithm 2 shows an implementation in pseudocode. The algorithm must be run for each entry in the non-RDF data and compared to each RDF term in the output RDF graph, and each RDF term must be compared to all protected IRIs.

Algorithm 2: Term Map Type Identification

Input: RDF term T (string), non-RDF input element E (string), protected IRIs A (array[string])
Output: The identified term map type (string)

```
// Remove percent encoded chars
T = decode(T);
// Remove protected IRI
for iri in A do
    if iri in E then
        T ← T.remove(iri);
        break;
// Identify term map type
if E not in T then
    return constant;
else if E == T then
    return reference;
else
    return template;
```

When Algorithm 2 processes the first RDF term of the running example, i.e. $T =$ `http://example.com/user_10`, and the first element of the first row of `file1.csv`, $E =$ `10`, it determines that the term map type is `template`, since the entry `10` can replace parts of the RDF term T. However, if the algorithm processes the second element of the first row of `file1.csv`, $E =$`835`, it determines that the term map type is `constant`, since the entry `835` cannot replace parts of the RDF term T and is independent of the input.

Identifying Term Maps The last step is to identify term maps. Term maps come in three variants: a template string, a reference identifier, and a constant string. In order to generate the correct term map, an algorithm must perform string substitution operations and check whether parts of the non-RDF source data are a valid substring of each RDF term in the output RDF graph, indicating a template or a reference. No match indicates a constant. Identifying the term map type uisng Algorithm 2 in advance helps to generate the term map. Algorithm 3 demonstrates how term maps can be generated.

Algorithm 3: Term Map Identification

Input: RDF term T (string), non-RDF input element E (string), RDF term type P (string)
Output: The identified term map (string)

```
// Remove percent encoded chars
T = decode(T);
// Generate Term Map
if P == 'constant' then
    return T;
else if P == 'reference' then
    return E;
else if P == 'template' then
    // Get attribute name of E
    E_Header ← attributeNameOf(E);
    // Replace E with attribute name of E
    T ← T.replace(E, E_Header);
    return T;
```

When Algorithm 3 is invoked with the first input from the example in Algorithm 2, i.e. $T =$ `http://example.com/user_10`, $E =$ `10`, $P =$ `template`, it produces the term map template string `http://example.com/user_{ID}`. Similarly, if the algorithm is invoked with the second input from the example in Algorithm 2, i.e. $T =$ `http://example.com/user_10`, $E =$ `835`, $P =$ `constant`, it produces the term map constant string `http://example.com/user_10`. Both are valid ways to generate the term map T, depending on the non-RDF input data, but the approach based on the template string is more general. When all elements of RDF triple, i.e., subject, predicate, and object, have been processed by algorithm 3, the extracted information can be used to fill a triple map template that generates exatly the given RDF triple. All triple maps generated in this way are added to the initial search space.

Complexity Analysis The Triple Map Generator component containing the three introduced algorithms is computationally very intensive and the performance depends on five factors, the number of processed non-RDF files r, the number of entries in each non-RDF file, n (for CSV files, this is the number of rows), the number of elements in each entry, m (for CSV files, this is the number of columns), the number of N-Triples in the output RDF graph, q, and the number of protected IRIs, p. The resulting overall time complexity is therefore $\mathcal{O}(r \cdot n \cdot m \cdot q \cdot p)$. Since r, m and p are relatively small compared to n and q, the time complexity of the component can be simplified to $\mathcal{O}(n \cdot q)$. The output of the Triple Map Generator component is the initial search space, represented in the form of a set of potential triple maps S.

4.2. Join Detector

Identifying join triple maps and corresponding join attributes, called `parent` and `child` in RML, is a challenging task. The Join Detector component of the reverse engineering pipeline takes the initial search space, i.e. the set of triple maps, S, generated by the Triple Map Generator component and identifies potential equi joins by comparing each generated triple map with all other triple maps. Once a potential join is identified, we use a set similarity overlap metric to heuristically identify join attributes and rank them by a score. We assume that among all potential join pairs, the pairs with the highest overlap are the most likely candidates for the join. Based on the extracted information, new join triple maps are generated and added to the set of all triple maps S, which extends the initial search space.

Join Detection To identify two triple maps that can potentially be combined into a single join triple map, we use Algorithm 4. The algorithm takes two triple maps, $tm1$ and $tm2$ as input, where

$tm1, tm2 \in T$, and determines whether they can be merged into a single join triple map. To do so, the algorithm compares the term maps, term map types, and term types of $tm1$ and $tm2$, as well as relevant invariants, which are defined as the longest common starting substring of RDF terms [10].

For $tm1$ to be considered a potential join triple map, its subject term map type must not be of term map type `constant`, while its object term map must be of term map type `constant`. Similarly, for $tm2$ to be considered a potential join triple map of $tm1$, the subject term map of $tm2$ must have a term map type of `constant`, while its object term map must not be of term map type `constant`. Additionally, both triple maps, $tm1$ and $tm2$, must have identical predicate term maps and predicate term types. Furthermore, the subject term map invariant of $tm1$ must be contained within the subject term map of $tm2$, and the object term map invariant of $tm2$ must be contained within the object term map of $tm1$.

If all these conditions are met, the two triple maps can potentially be combined into a single join triple map. The following Algorithm 4 presents an implementation of the described join triple map identification.

Algorithm 4: Join Triple Map Identification

Input: Triple Map 1 $tm1$ (graph), Triple Map 2 $tm2$ (graph)
Output: combination possible (boolean)

```
// tm1 must not be constant in subject and must be constant in object
if tm1.subject_term_map_type == 'constant' then
    return false;
if tm1.object_term_map_type != 'constant' then
    return false;
// tm2 must be constant in subject and must not be constant in object
if tm2.subject_term_type != 'constant' then
    return false;
if tm2.subject_term_type == 'constant' then
    return false;
// tm1 and tm2 must have equal predicate term map and term map type
if not (tm1.predicate_term_map == tm2.predicate_term_map and
  tm1.predicate_term_type == tm2.prediacte_term_type) then
    return false;
// Invariant of tm1 subject must be in tm2 subject
if not( invar(tm1.subject_term_map) in tm2.subject_term_map) then
    return false;
// Invariant of tm2 object must be in tm1 object
if not( invar(tm2.object_term_map) in tm1.object_term_map) then
    return false;
return true;
```

The input to the Join Identification component is the initial search space, represented by the set S. The set S contains two triple maps, $tm1$ and $tm2$, which are depicted in Fig. 3. If both triple maps are used as input for Algorithm 4, all checks pass, and the output is `true`, indicating that both triple maps potentially form a join.

Identifying Join Attributes Once two potential join triple maps have been identified, the join attributes must be determined. Our approach employs a heuristic that generates a score for each possible combination of join attributes, using set similarity measures, and compares the generated RDF triple with the expected RDF data to identify the best pair. The method is similar to established

tm1

```
1. @base <http://ex.com/base/> .
2.
3. <#TriplesMap98> a rml:TriplesMap ;
4.   rml:logicalSource [ rml:source [ rml:path "file1.csv"];
5.   rml:predicateObjectMap [
6.    rml:object <http://example.com/legal> ;
7.    rml:predicate <http://ex.com/worksAt> ] ;
8.    rml:subjectMap [ rml:template "http://ex.com/user_{ID}"] .
```

tm2

```
1. @base <http://ex.com/base/> .
2.
3. <#TriplesMap89> a rml:TriplesMap ;
4.   rml:logicalSource [ rml:source [ rml:path "file2.csv"];
5.   rml:predicateObjectMap [
6.    rml:objectMap [ rml:template "http://ex.com/{Name}"] ;
7.    rml:predicate <http://ex.com/worksAt> ] ;
8.    rml:subject <http://ex.com/user_10> .
```

Figure 3: Two triple maps, $tm1$ and $tm2$, contained in the initial search space, represented by the set S.

approaches used to identify joinable tables in large-scale data lakes [24].

To calculate the score, the algorithm evaluates the overlap coefficient, also known as set containment [25], which measures the similarity between two finite sets, A and B. The overlap coefficient is computed using Equation 1. The resulting values range from 0 to 1, i.e., $0 \leq overlap(A, B) \leq 1$, where the value 1 is reached if $A \subseteq B$ or $B \subseteq A$.

$$overlap(A, B) = \frac{|A \cap B|}{min(|A|, |B|)} \tag{1}$$

Once the highest ranking join pairs have been identified, the input is joined and the pair that results in the generation of the most RDF triples in the output is selected and used as `parent` and `child` attributes in the generated triples map.

The overlap coefficient of the data used in the two triple maps, $tm1$ and $tm2$, in the running example and Fig. 3 is highest for the combination `(Department, ID)`, as all other combinations have no overlap. Therefore, both triple maps are combined into a single join triple map with `Department` as child and `ID` as parent. The new join triple map is then added to the search space, represented by the set S.

Complexity Analysis The Join Identification component consists of two steps, the Join Detection step and the Join Attribute Identification step. The first step, which is also the most significant in terms of time complexity, involves Algorithm 4, which must be executed for all generated triple maps in the search space, comparing each one with all other generated triple maps. Consequently, the Join Detection step has a quadratic time complexity, dependent on the size of the initial search space, represented by the generated triple map set, S. In contrast, the Join Attribute Identification step iterates over the content of the two compared attributes and, therefore, has a linear time complexity. Thus, the overall time complexity of the Join Identification component depends on the size of S, given by $\texttt{size}(S)$, resulting in a time complexity of $\mathcal{O}(\texttt{size}(S)^2)$.

4.3. Triple Map Filter

The final component in the reverse engineering pipeline is the Triple Map Filter component, which first identifies and removes redundant triple maps, i.e., those triple maps that generate only a subset of RDF triples produced by other triple maps, from the set S. In the second step, the remaining triple maps in S are verified to ensure they generate the expected RML triples before being written to an output RML mapping document.

Filter Triple Maps Generating Common Subsets All triple maps in S produce RDF triples that are contained in the reference RDF output. However, not all triple maps generate unique RDF triples, some may produce a subset of RDF triples that can already be generated by another triple map. This leads to the generation of duplicate RDF triples, which increases computational overhead for RML interpreters. Therefore, the goal of this step is to identify triple maps that generate subsets of RDF triples already produced by another triple map and to filter out unnecessary triple maps from S. This ensures the generation of a minimal RML mapping document containing only the required triple maps.

To remove unnecessary triple maps from S, the Triple Map Filter component generates the RDF triples described by each triple map and compares them with all other RDF triples to identify subsets. If a triple map produces only a subset of another triple maps RDF triples, the triple map generating duplicates is removed from S.

For instance, looking back at the example used in combination with Algorithm 3, where two term maps have been identified. T_1 = `http://example.com/user_{ID}` with term map type `template` and T_2 = `http://example.com/user_10` with term map type `constant`. When processing T_1 and T_2 in combination with `file1.csv` of the running example, we can see, that T_1 produces two RDF output triple, one where the placeholder `ID` is replaced with `10`, i.e., `http://example.com/user_10`, and one where the palceholder is replaced with `20`, i.e., `http://example.com/user_20`. On the other hand T_2, only produces a constant output `http://example.com/user_10`. Therefore, T_1 is more general, and T_2 prodcues only a subset of RDF triple T_1 prodcues, meaning T_2 will be filtered out and only T_1 will be kept.

Validating Generated RDF Terms In the final step, the remaining triple maps in S are validated to ensure that all generated RDF terms are included in the output RDF graph. A triple map is considered correct and added to the resulting RML mapping document only if all the RDF terms it produces are present in the output RDF graph. Once validation is complete, the final set of triple maps, S, is written to disk, marking the completion of the generation process.

Complexity Analysis The final step of the reverse engineering pipeline first generates all RDF triples described in each generated RML triple map by processing all non-RDF input files, iterating over each entry in the non-RDF files. In a second step, the results are filtered by comparing the generated RDF triple identification subsets, and validated by comparing the generated RDF triple with the expected RDF triple in the output graph. However, the important step for the time complexity is the generation of the RDF triple, which depends on the number of triple maps t, the number of input non-RDF files r, and the number of entries in the non-RDF files n, resulting in a time complexity of $\mathcal{O}(t \cdot r \cdot n)$.

5. Evaluation

To evaluate the functionality of our approach, we implemented all of the introduced algorithms in a proof-of-concept and used the prototype implementation to generate RML mapping documents based on the latest version of the RML Core test cases[2]. The RML test cases were developed to validate the conformance of different RML interpreters to the RML specification [26].

5.1. Prototypical Implementation

To validate our RML mapping document generation approach, we developed a proof-of-concept implementation called the Reverse Engineering Mapping tool (`ReMap`), written in Python. The `ReMap` tool is based on the pandas[3] library for handling non-RDF input data and the RDFLib[4] library, combined with a custom N-Triples parser, to process RDF output data.

[2] https://github.com/kg-construct/rml-core/tree/980ca117443ae61ca6d72c0f2ba38967e4360c32
[3] https://pandas.pydata.org/
[4] https://rdflib.dev/

The ReMap tool is released under an open-source license and is publicly available on GitHub[5]. Our prototype implementation currently supports only CSV data as non-RDF input and N-Triple format for RDF output, as both formats are straightforward to parse. This allowed us to focus on developing the core reverse engineering functionality.

For easy distribution of the ReMap tool, we compiled the source code using Nuitka[6] into an executable to allow execution without a Python interpreter installed.

5.2. Conformance Evaluation of ReMap

To assess the conformance to the RML specification of the automatically generated RML mappings produced by our approach, we utilized the RML core test case dataset. We generated RML mapping documents for each test cases using our ReMap tool and analyzed the results.

Dataset We used the RML Core test cases dataset, limited to CSV input data, for our conformance evaluation. Each test case consists of one or more CSV files representing the non-RDF input data, a corresponding RML mapping document, and an expected RDF output file in N-Triples format. Out of the 48 available CSV test cases, we considered 35. The remaining 13 test cases were not applicable, as they evaluate the handling of errors in either the RML mapping document or the source data.

Execution Process We iterated through all the test cases with ReMap, using the provided CSV files and the expected RDF graph as input. We then performed the mapping reverse engineering step and saved the generated RML mapping document for each test case.

Next, we set up BURP [27] v0.1.1, an RML interpreter designed for compliance with the RML specification [28] rather than execution speed or memory efficiency. We executed our generated RML mapping documents with BURP and compared the resulting RDF output with the original expected RDF data to verify that the generated RML mappings produced equivalent RDF data.

Two test cases (RMLTC0024e-CSV and RMLTC0024f-CSV) could not be validated using BURP, as the required RML constructs are not yet supported. Therefore, we manually compared the generated RML mapping documents for these cases.

Results The ReMap tool was able to generate an equivalent RML mapping document for all test cases, demonstrating that our approach is viable.

5.3. Conformance Evaluation using a LLM

The current generation of LLMs can also generate RML mapping documents based on a given non-RDF input and RDF output. For our evaluation, we used OpenAI's best available reasoning model, o3-mini-high. The model has a knowledge cutoff in October 2023 and, therefore, only supports the older RML_{IO} vocabulary. All prompts used, as well as the generated mappings, are publicly available on GitHub[7].

Dataset We used the same 35 RML core test cases as those in the previous section (see Section 5.2).

Execution Process For each test case, we provided the input CSV data along with the expected RDF output graph in a simple prompt, instructing the LLM to generate a corresponding RML mapping document. The model then generated an RML mapping document, which we used to execute the mapping using RMLMapper v7.3.1[8]. Finally, we compared the generated RML output graph to the expected RDF graph.

[5] https://github.com/FreuMi/remap
[6] https://nuitka.net/
[7] https://github.com/FreuMi/remap/tree/main/llm_test_cases
[8] https://github.com/RMLio/rmlmapper-java/releases/tag/v7.3.1

Results The LLM successfully generated the correct RML mapping document for 22 test cases, but failed for 13, resulting in a 63% success rate, which is 31% worse than the `ReMap`. The 13 failed test cases were primarily the more complex ones, suggesting that LLMs can handle simpler cases effectively, but struggle with more complicated ones. Additionally, the model required extensive processing time, ranging from 10 seconds for simpler test cases to up to 70 seconds for more complex ones.

5.4. Performance Evaluation

To empirically assess the RML mapping document generation speed and memory consumption of the developed `ReMap` tool we ran experiments on on a virtual machine running on an Intel Xeon Gold 6154 CPU. The virtual machine has access to 8 cores running at `3.0 GHz` each, `64 GB` of RAM and an `8 GB` swap partition. The system is running Ubuntu 24.04.1 LTS and the Python scripts are executed using the Python 3.12.3 interpreter. Execution time and peak memory usage were measured using the `time` command from the GNU time package[9].

Dataset We reuse the `duplicated values` dataset from the benchmark used in the KG Construction Workshop (KGCW) Challenge 2024 [29]. We adapted the included RML mapping document to directly process the CSV data. The `duplicated values` dataset contains synthetic data specifically designed to evaluate RML interpreters and their handling of duplicate values.

Execution Process For our experiments, we reduced the number of rows in the CSV data, m, to 5, 10, 20, and 30 unique rows, which correspond to 100, 200, 400, and 600 N-Triples, q, in the RDF output graph. All other parameters were kept constant, meaning the number of CSV input files, r, was set to 1, the number of columns in each CSV file, m, remained at 21, and the number of protected IRIs, p, was fixed at 3.

We used Morph-KGC [10], along with the original RML_{IO} v1.1.1[10]-based mapping document and the reduced CSV data, to generate the expected RDF graph. Next, we employed `ReMap` to reverse engineer an equivalent RML mapping document based on the RDF graph produced by Morph-KGC and the reduced CSV files. The generated RML mapping document was then processed using BURP [27], and the resulting RDF output was compared to the original RDF output to validate correctness.

Results Fig. 4 plots the benchmark results, showing an exponential increase in execution time coupled with a similar increase in memory consumption. The observed results are consistent with expectations, as the time complexity analysis of the algorithms predicted such exponential growth. We also found that all experiments generated RML mapping documents that produced the expected RDF graph based on the non-RDF input data.

Overall, the results indicate that while the `ReMap` prototype implementation is not suitable for large datasets due to long processing times and high memory consumption, it performs reasonably well for smaller datasets typically used to generate RML mapping documents based on examples.

The performance of the `ReMap` tool is limited because it does not utilize multiprocessing and, therefore, runs on a single core. Additionally, the high number of string replacement and manipulation operations, currently executed in pure Python, are time-consuming. This is due to Python's string immutability[11], which requires the time-consuming creation of a new string object each time a modification is made.

Additionally the experiment demonstrated, that the `ReMap` tool can be used to update older mapping documents using the RML_{IO} standard to the latest RML standard. Since when older RML mappings are executed and a RDF output graph is generated, `ReMap` can use the original non-RDF input data and the generated RDF ouput, to reverse engineer a RML mapping document based on the newest standard.

[9] https://www.gnu.org/software/time/
[10] https://rml.io/specs/rml/v/1.1.1/
[11] https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str

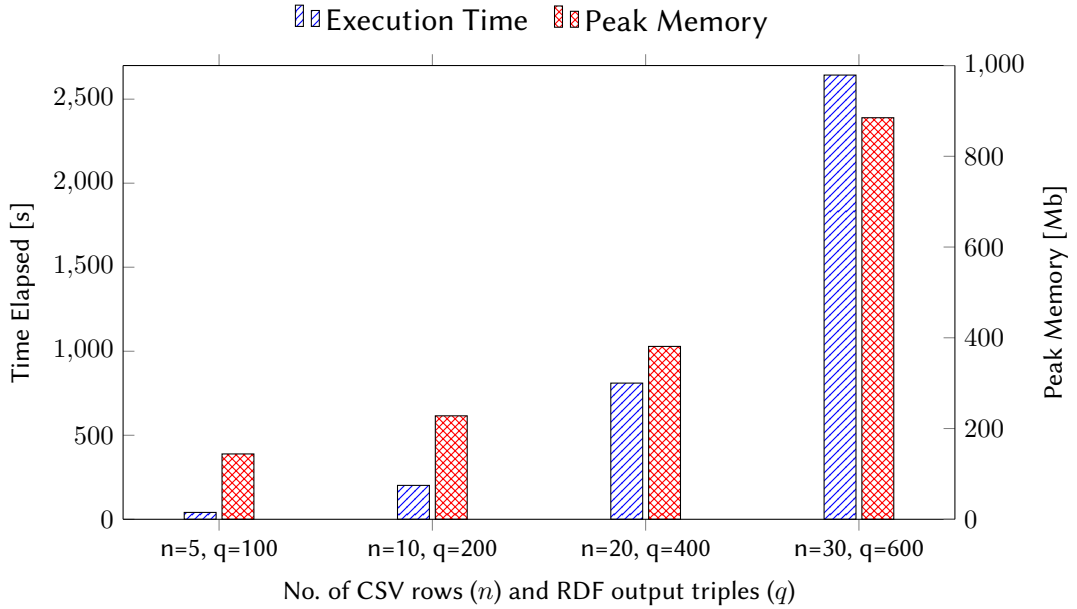

Figure 4: Average execution time and memory consumption of ReMap generating RML mapping documents over 3 runs with different input sizes.

6. Conclusion and Future Work

We introduced an RML mapping document reverse engineering approach capable of generating a corresponding RML mapping document based on given non-RDF source data and an expected output RDF graph. We presented and described the core algorithms required to implement the reverse engineering pipeline and discussed their time complexity.

Additionally, we introduced a prototype implementation called ReMap. ReMap enables users to define only the expected output RDF graph and provide the non-RDF source input to automatically generate an RML mapping document. Additionally, since the reverse engineered RML mapping document does not rely on the original RML mapping document, ReMap can be used to update older mappings using the RML_{IO} to the latest RML standard using the latest vocabulary. We used the ReMap implementation to evaluate the conformance of the generated RML mapping documents using the RML core test cases. Furthermore, we assessed the performance of ReMap, measuring execution time and memory consumption using an established benchmark dataset from the KGCW Challenge. Our evaluation showed that the proposed approach conforms to all applicable RML core test cases, since the ReMap tool generates equivalent RDF mapping documents for all test cases. The empirical analysis of execution time and memory consumption demonstrated exponential growth, which aligns with expectations based on the time complexity of the involved algorithms.

For future work, on the theoretical side, we aim to improve the formalization of the RML mapping reverse engineering approach using set notation. On the application side, we plan to integrate the proposed reverse engineering pipeline into a user-friendly GUI-based application. A GUI-based application will allow users to define the mapping of non-RDF data to RDF by example, and simplify the updating of RML mapping documents. By providing a easy-to-use tool for the mapping by example process, the application will reduce the need to manually write or rewrite RML mapping documents from scratch.

Acknowledgments

This work was partially funded by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) through the Antrieb 4.0 project (Grant No. 13IK015B).

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT 4o to check grammar and spelling.

References

[1] R. Cyganiak, D. Wood, M. Lanthaler, RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation, 2014. URL: https://www.w3.org/TR/rdf11-concepts/.

[2] A. Hogan, E. Blomqvist, M. Cochez, et al., Knowledge Graphs, ACM Comput. Surv. 54 (2021).

[3] A. Dimou, M. Vander Sande, P. Colpaert, et al., RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data, Ldow 1184 (2014).

[4] P. Heyvaert, A. Dimou, A.-L. Herregodts, et al., RMLEditor: A Graph-Based Mapping Editor for Linked Data Mappings, in: Proceedings of the 13th Extended Semantic Web Conference, Springer, 2016, pp. 709–723.

[5] P. Heyvaert, B. De Meester, A. Dimou, R. Verborgh, Declarative rules for linked data generation at your fingertips!, in: The Semantic Web: ESWC 2018 Satellite Events: ESWC 2018 Satellite Events, Heraklion, Crete, Greece, June 3-7, 2018, Revised Selected Papers 15, Springer, 2018, pp. 213–217.

[6] D. Van Assche, T. Delva, P. Heyvaert, B. De Meester, A. Dimou, Towards a more human-friendly knowledge graph generation & publication, in: ISWC2021, The International Semantic Web Conference, volume 2980, CEUR, 2021.

[7] A. Iglesias-Molina, D. Van Assche, J. Arenas-Guerrero, et al., The RML Ontology: A Community-Driven Modular Redesign After a Decade of Experience in Mapping Heterogeneous Data to RDF, in: International Semantic Web Conference, Springer, 2023, pp. 152–175.

[8] P. Grossi García, Automatic generation of R2RML and RML mappings, Master's thesis, Universidad Politécnica de Madrid, ETSI Informatica, 2022.

[9] C. Allocca, A. Gougousis, A Preliminary Investigation of Reversing RML: From an RDF dataset to its Column-Based data source, Biodiversity data journal (2015).

[10] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, et al., Morph-KGC: Scalable knowledge graph materialization with mapping partitions, Semantic Web 15 (2024) 1–20.

[11] E. Iglesias, S. Jozashoori, D. Chaves-Fraga, et al., SDM-RDFizer: An RML Interpreter for the Efficient Creation of RDF Knowledge Graphs, in: Proceedings of the 29th ACM international conference on Information & Knowledge Management, 2020, pp. 3039–3046.

[12] S. M. Oo, G. Haesendonck, B. De Meester, et al., RMLStreamer-SISO: An RDF Stream Generator from Streaming Heterogeneous Data, in: International Semantic Web Conference, Springer, 2022, pp. 697–713.

[13] M. Freund, S. Schmid, R. Dorsch, et al., FlexRML: A Flexible and Memory Efficient Knowledge Graph Materializer, in: Proceedings of the 21st Extended Semantic Web Conference, Springer, 2024, pp. 40–56.

[14] M. Scrocca, A. Carenini, M. Grassi, M. Comerio, I. Celino, Not Everybody Speaks RDF: Knowledge Conversion between Different Data Representations, in: Fifth International Workshop on Knowledge Graph Construction@ ESWC2024, 2024.

[15] Z. Zhang, Effective and efficient semantic table interpretation using tableminer, Semantic Web 8 (2017) 921–957.

[16] U. Khurana, S. Galhotra, Semantic concept annotation for tabular data, in: Proceedings of the 30th ACM International Conference on Information & Knowledge Management, 2021, pp. 844–853.

[17] T. Mesnard, C. Hardin, R. Dadashi, et al., Gemma: Open models based on gemini research and technology, arXiv preprint arXiv:2403.08295 (2024).

[18] B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, et al., Language models are few-shot learners, arXiv preprint arXiv:2005.14165 1 (2020).

[19] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, S. Zhong, B. Yin, X. Hu, Harnessing the power

of llms in practice: A survey on chatgpt and beyond, ACM Transactions on Knowledge Discovery from Data 18 (2024) 1–32.
[20] M. Val-Calvo, M. E. Aranguren, J. Mulero-Hernández, G. Almagro-Hernández, P. Deshmukh, J. A. Bernabé-Díaz, P. Espinoza-Arias, J. L. Sánchez-Fernández, J. Mueller, J. T. Fernández-Breis, OntoGenix: Leveraging Large Language Models for enhanced ontology engineering from datasets, Information Processing & Management 62 (2025) 104042.
[21] M. Hofer, J. Frey, E. Rahm, Towards self-configuring knowledge graph construction pipelines using llms-a case study with rml, in: Fifth International Workshop on Knowledge Graph Construction@ ESWC2024, 2024.
[22] Q. T. Tran, C.-Y. Chan, S. Parthasarathy, Query reverse engineering, The VLDB Journal 23 (2014) 721–746.
[23] M. Arenas, G. I. Diaz, E. V. Kostylev, Reverse engineering SPARQL queries, in: Proceedings of the 25th international conference on world wide web, 2016, pp. 239–249.
[24] E. Zhu, D. Deng, F. Nargesian, R. J. Miller, Josie: Overlap set similarity search for finding joinable tables in data lakes, in: Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 847–864.
[25] E. Zhu, F. Nargesian, K. Q. Pu, R. J. Miller, LSH ensemble: Internet-scale domain search, arXiv preprint arXiv:1603.07410 (2016).
[26] P. Heyvaert, D. Chaves-Fraga, F. Priyatna, et al., Conformance Test Cases for the RDF Mapping Language (RML), in: Iberoamerican Knowledge Graphs and Semantic Web Conference, Springer, 2019, pp. 162–173.
[27] C. Debruyne, D. Van Assche, A Fresh Start: Implementing an RML Processor from Scratch to Validate RML Specifications and Test Cases, in: Fifth International Workshop on Knowledge Graph Construction@ ESWC2024, 2024.
[28] D. Van Assche, C. Debruyne, BURPing Through RML Test Cases, in: Fifth International Workshop on Knowledge Graph Construction@ ESWC2024, 2024.
[29] D. Van Assche, D. Chaves-Fraga, A. Dimou, et al., KGCW 2024 Challenge, 2024. URL: https://doi.org/10.5281/zenodo.11577087.