GRAPE: Guiding RML Authoring with a Projectional Editor

Jakub Duchateau, Christophe Debruyne

Montefiore Institute, University of Liège, Belgium

Abstract

KG construction often involves mapping data from various sources to RDF, which can be achieved with declarative languages such as RML. RML, while powerful, presents authoring difficulties due to its RDF graph-based structure and Turtle serialization. Tools and languages have been proposed to help the creation of mappings by creating (UI) abstraction of RML concepts or more accessible languages. However, existing tools often struggle to balance guidance and flexibility. While representing RML in RDF allows enriching mappings with other vocabularies, it also presents a steep learning curve for users unfamiliar with Turtle and RML. This paper introduces GRAPE, an open-source projectional editor designed to facilitate RML mapping creation while preserving the flexibility of RDF schemas used to extend RML. GRAPE uses a language-oriented approach, treating each RML module as a domain-specific language. This allows users to write RML and/or Turtle in one artifact. Implemented using JetBrains MPS, GRAPE provides a text-like projection for manipulating the abstract syntax tree. GRAPE ensures syntactic correctness, prevents errors and guides with auto-completion and intentions. This lowers the barrier for new users while preserving the flexibility required for complex mappings. This paper aims to demonstrate the feasibility of this approach. It also describes the steps we have conducted to create an editor for Turtle, which is then extended for RML and its modules. The approach is extensible and allows the community to develop languages for other vocabularies.

Git: https://gitlab.uliege.be/JakubDuchateau/GRAPE

Tutorial and videos: https://jakubduchateau.gitlabpages.uliege.be/grape/

Keywords

RML Editor, Knowledge Graph Construction, Projectional Editor

1. Introduction

Constructing RDF Knowledge Graphs is challenging, as one needs to map various non-RDF sources to RDF, often according to a vocabulary or ontology for which the correspondence is not trivial. Declarative approaches, which allow one to describe how data sources should be mapped onto RDF using a special language, have emerged to address this problem. A special processor then processes these mappings, and the processor is responsible for "executing" the mapping. The RDF Mapping Language (RML) [1], which aims to be a candidate for W3C Recommendation, is one such language. However, RML comes with its challenges, especially when authoring the mappings.

RML is stored as an RDF graph. Moreover, at the time of writing, the current RML specification states that RML mappings are stored in the Turtle serialization format. Nevertheless, authoring RML (in Turtle) is difficult. The community has looked at various approaches to guide and facilitate mapping authoring to address this difficulty. The problem is that many of these approaches do not provide the freedom that RDF graphs offer, as we will describe in Section 2. We suspect that more technical profiles, such as developers accustomed to technologies other than RDF or RML, will find existing RML editing techniques either too constrained or with a steep learning curve, requiring a deep understanding of the specifications before they can be effectively used.

This work aims to develop a tool that balances guiding users in creating well-formed RDF documents and RML mappings and retaining the flexibility of authoring RDF in a text-like editor. Such a tool would achieve what is called a *a low threshold and high ceiling* [2], which means that *"simple [mappings] should be simple, and complex [mappings] should be possible."* [3] Such a tool should guide through features

KGCW'25: 6th International Workshop on Knowledge Graph Construction, June 1 or 2, 2025, Portorož, Slovenia

Jakub.Duchateau@uliege.be (J. Duchateau); C.Debruyne@uliege.be (C. Debruyne)

0009-0009-5090-8192 (J. Duchateau); 0000-0003-4734-3847 (C. Debruyne)

© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

such as syntax error prevention, auto-completion, and suggestions, thereby lowering the threshold for users. Simultaneously, it should maintain flexibility by allowing the interlacing of *free-form RDF*[1] and RML. Suppose one wants to obtain the flexibility of authoring RDF and RML in a text editor. In that case, a tool should be able to switch context, i.e., handle different contexts such as RML or free-form RDF in one document. If this is possible, then the runtime composition of new custom concepts ensures that the ceiling is not limited. Additionally, it attempts to offer a keyboard-driven experience that is not too dissimilar to textual programming, improving familiarity for developer users.

The problem of parsing different languages in one text can be avoided with *projectional editing*, and this is the solution that we will adopt and explore in this study. Projectional editing is an alternative to traditional parsing approaches, where the user interacts directly with the Abstract Syntax Tree (AST) structures.[2] This approach eliminates the need for parsing and ensures syntactic correctness, making it particularly advantageous for composing languages and concrete syntaxes. Projectional editing is a well-established paradigm for Domain-Specific Languages (DSLs) [5], as exemplified in [6] where C is implemented in MPS. This approach provides a shorter feedback loop because changes are immediately reflected in the AST, enabling instant validation.

In short, we want to explore whether it is feasible to propose mapping authors who are more technically savvy with a tool that approaches a text-based editing approach, providing all the support for writing well-formed RML (e.g., autocompletion) whilst providing them the liberty of turtle-based editing. The contributions of this paper are an approach and prototype implementation for creating and authoring RML mappings in such a manner. Our study demonstrates the feasibility of language-oriented programming through a projectional editor for RML, treating each part of RML as a domain-specific language while keeping the extensibility of the turtle notation or your custom-specific notation.

The editor, GRAPE, is made available as an open-source prototype implementation. GRAPE can process existing RML mappings, i.e., one does not have to create mappings in GRAPE from scratch, and the RML generated by GRAPE can be given to any RML-core compliant RML processor.

2. Related work

Various tools and methodologies have been proposed for RML authoring and related declarative languages for data integration and data integration mappings. We refer to [7, 8] for a review. In this section, we will describe some common approaches. This section focuses on editors for KG generation.

There are editors with a strong visual component, e.g., based on graphs and trees. For example, RML Editor [9] offers a visual editor with node-link diagram graph visualization and a point-and-click experience. JUMA [10] provides a jigsaw-based editor for (R2)RML where users are guided in creating well-formed mappings where blocks follow the structure of the mapping languages.

Then, there are text-based approaches. YARRRML uses YAML as a serialization format, providing a textual abstraction of RML concepts. Yatter [8] allows one to transform YARRRML mappings in RML (as Turtle) and vice versa. Where YARRRML is the language, which can be authored with a text editor, Matey [11] is a web-based integrated development environment (IDE) for YARRRML, featuring syntax highlighting and panels for sources, mappings, and outputs. Another example of a text-like approach is XRM[3]. XRM is a DSL for (R2)RML implemented with XText[4], offering a textual syntax and editor for RML with auto-completion. The authoring of RML in RDF serializations allows one to include statements with other vocabularies (e.g., metadata with PROV-O). XRM only provides a DSL for (R2)RML. The XText language workbench relies on a parser, which means that considering multiple

[1]Free-form RDF refers to RDF that is not necessarily related to RML. Examples could include adding metadata in PROV-O[4] to a triples map.

[2]In text-based editing, one manipulates a text document that is subsequently parsed by some software. For example, one authors an RML mapping in VS Code, parsed and executed with an RML Engine. With projectional editing, you interact directly with the document's underlying Abstract Syntax Tree (AST). Since one is working with the AST directly, the computer does not need to parse the text to understand its structure.

[3]XRM: https://zazuko.com/products/expressive-rdf-mapper/

[4]https://eclipse.dev/Xtext/

languages requires a grammar that includes all of those.

Other approaches include guided workflows (also called "wizards") with RMLx [12], programming-by-example approaches with Karma [13], and spreadsheet-based tools like Mapeathor [14]. Each has varying degrees of freedom and usability.

From the observed editors, we identify a trade-off between guidance and flexibility. Visual editors offer strong guidance but often at the expense of liberty, making complex mappings difficult or even impossible. For example, in JUMA, one must start with puzzles that resemble triple maps, where one needs to provide the pieces for logical sources, subject maps, predicate object maps, etc., in a set order and create tree-like mappings. One cannot reuse predicate object maps across mappings or declare those outside a triples map. The tools ensure the production of well-formed (R2)RML but do not allow the flexibility offered by RDF. Furthermore, it can not include additional metadata with other vocabulary.

In contrast, authoring RML mappings with textual editors provides greater liberty. RDF is a graph-based data model, but it is often serialized and manipulated in textual formats such as Turtle. This textual nature of these serialization formats naturally lends itself to text-based editing, but this comes with a steep learning curve and makes them less accessible to those not proficient in Semantic Web technologies.

This contrast highlights a gap in existing tools: an editor that combines supporting the users through features like autocompletion, which relies on recognition rather than recall while maintaining the flexibility required for complex RML mappings in a text editor or a text-editor-like environment.

In the next section, we present our approach, which introduces adapted notations and a balance between guidance and freedom to improve the RML authoring experience.

3. Towards Projectional Editors for RML

RML can be considered a *Domain-Specific Language* (DSL) for KG Generation. A DSL is *"a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."* [15] As such, we could approach RML as a DSL by adopting DSL design techniques. However, RML is often authored in RDF serializations. RDF is special as it is a flexible model that allows us to use different vocabularies. Its graph structure also allows us to represent and combine information arbitrarily. One could thus consider that RDF allows us to mix languages if each vocabulary is considered to be a "DSL". As different languages need to be supported, we must find a way to ensure that the document being authored can still be parsed.

The RML specification offers ontologies and shapes formalizing well-formed and valid mappings. We will follow a Language-Oriented Programming paradigm to create a DSL informed by the RML specification and the Turtle serialization format. The reason is that a DSL for Turtle will allow us to extend it with DSLs for other vocabularies such as RML. Our approach will consider each RML module a domain-specific language with a specialized concrete syntax that can be mixed within an RDF document.

3.1. Projectional Editing

Suppose one wants to obtain the flexibility of authoring RML and RDF in a text editor. In that case, a tool should be able to switch context, i.e., handle different contexts such as RML or free-form RDF. Switching contexts is hard, and addressing this problem while parsing is complex. One can address this problem by avoiding the parsing process with a technique called *projectional editing*. Projectional editing, or structural editing, offers an alternative to traditional approaches that rely on parsing. Instead of parsing user-written text in a concrete syntax to derive an Abstract Syntax Tree (AST), projectional editors allow users to directly interact with the AST structures. User actions are immediately reflected in the structure of the AST, bypassing the need for a parsing step.

To achieve this, we must support two perspectives. First, the editor must be able to represent any additional RDF Node within the mappings. Second, by approaching each vocabulary as a language, the

editor should be able to mix languages within a single "document"[5] to form an RDF graph. Support of the former will allow us to write arbitrary RDF and thus use any vocabulary. Still, when patterns (concerning a vocabulary or even a combination of vocabularies) emerge, one should consider creating a DSL. This approach can also be adopted for vocabularies consisting of modules, such as RML, and we should enable and facilitate the composition of these modules.

Projectional editing inherently supports unparseable code since it does not rely on grammar or parsing. This eliminates grammar ambiguities, making it particularly advantageous for composing languages and syntaxes [16], which fits the language-oriented approach we want to adopt.

While this paper focuses on a textual view with a keyboard-driven editing experience to study the familiarity of source code editing for developers, other visual forms of editing are also possible. However, projectional editors behave differently to textual notations from what developers know from traditional editors for parser-based languages [17], which may need some adaptations of users. For example, authors must indicate, in some way, the DSL they want to use in a particular part of the document.

To help in the process of building an editor for a language, Language Workbenches (LW) [18] are a class of tools that facilitate the implementation of DSLs. Using a language workbench for RML has already been used successfully with XRM. We emphasize the compositional aspect here, closely following the RML specifications so that existing knowledge is transferable while additional abstractions could be added above it.

3.2. Background: JetBrains MPS

We use JetBrains Meta Programming System (MPS)[6] to formalize the Turtle and RML language structures and to facilitate the creation of editor interfaces with an IDE infrastructure.

Briefly, MPS is a Language Workbench[18], where *Structures* define the concepts of a language, representing the Abstract Syntax Tree (AST). These structures, similar to classes in object-oriented languages like Java, implement inheritance, composition and allowing us to declare properties, children and references. These allow us, for example, to declare a concept "Triples Map" having zero or more "Predicate Object Maps" as children. Other modules, also called aspects of a concept, provide functionality for editor interaction, type systems, scoping, and (text) generation.

Constraints are dynamic rules that indicate what concept to use when a new node is created, limiting the AST that can be formulated. These dynamic constraints guide users in writing statements, complementing the static constraints encoded within the language's structure. For example, such aspects ensure that term maps refer to the correct term type.

Editors in MPS are described with a cell system that forms a textual view with cells that are modifiable define how the AST is viewed and edited. These cells can be arranged horizontally, vertically, or indented to mimic blocks of text, but the editors can also be adapted for mathematical notation, graphs, diagrams, and tables.

Generators in MPS facilitate model-to-model transformations, generally converting higher-level language models to lower-level ones. In our case, this involves transforming RML concepts into Turtle concepts. A special kind of generator, *textGen*, specialises in generating textual representations from models, specifically it will convert Turtle concepts into their textual syntax. MPS also integrates various IDE features such as running and debugging code, version control, code completion, and type systems, which can be leveraged to enhance the development and editing experience.

The abstract ideas introduced in this section will be illustrated with concrete examples for the projectional editing of Turtle and RML in the next section.

[5]We use quotes as the authors will engage with something that looks and behaves like text, but the artifact contains a representation of the various ASTs. We will use "textual view" and "document" interchangeably when talking about the projectional text editor.

[6]https://www.jetbrains.com/mps/

4. GRAPE: A guiding editor ecosystem for RML

GRAPE is an open-source projectional editor prototype for RML (and Turtle), developed with JetBrains MPS. It formalizes Turtle and RML specifications as concepts in composable domain-specific languages guided by a language-oriented paradigm. It provides fundamental IDE functionalities such as version control, running code integration, and viewing standard data files. The choice of MPS is also motivated by the benefits of projectional editing, which offers guidance, prevents syntactic errors and enables composition, while having reasonable text-like projection.

4.1. Turtle in GRAPE

Given the necessity of an RDF-compatible language, both as a generation target for exporting and to be able to mix any RDF nodes to extend non-standard mapping, we follow RML specifications by adopting Turtle as a base language.

Turtle concepts in GRAPE are based on the Turtle 1.1 grammar [19]. All Turtle 1.1 constructs are supported, and some Turtle 1.2 [20] features, such as the annotation block, are also included. We base ourselves on those grammars but do not follow them strictly. When necessary, we created additional abstractions to support extensions by other DSLs.

To illustrate our approach, we show how we transformed a Turtle grammar rule into structures and aspects in MPS. We introduced the *SubjectTriples* concept in our Turtle language that represents a *subject* associated with a *list of predicate objects*, as shown in Figure 1.

```
concept SubjectTriples extends    TriplesStatment
                       implements Triples

  instance can be root: false
  alias: subject (verb object)* .
  short description: <no short description>

  properties:
  << ... >>

  children:
  subject              : Subject[1]
  predicateObjectList  : PredicateObjectList[1]

  references:
  << ... >>
```

(a) SubjectTriples structure aspect.

```
<default> editor for concept SubjectTriples
  node cell layout:
    [-
    % subject % % predicateObjectList %
      . /folded cell: [- % subject % with * R/O model access * predicates . -]
    -]

  inspected cell layout:
    <choose cell model>
```

(b) SubjectTriples editor is declared in the editor aspect with the subject followed by predicate objects indented. When folded, it shows the subject and the number of predicate objects only.

```
text gen component for concept SubjectTriples {
  (node)->void {
    append ${node.subject};
    with indent {
      append ${node.predicateObjectList};
    }
    append \n {.} \n;
  }
}
```

(c) SubjectTriples textGen aspect.

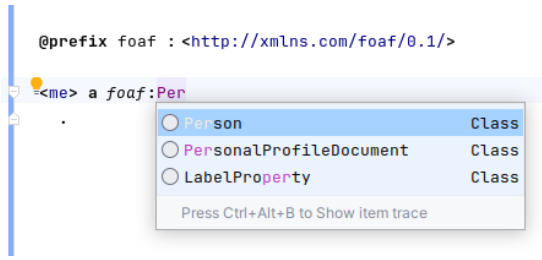
Figure 1: SubjectTriples is a concept in Turtle language that represents Turtle triples.

The editor aspect "simulates" a Turtle document by showing the concepts in their textual representation. Due to the projectional nature of the editor, new lines and alignments follow what is decided in the editor aspect. Alternatives could be implemented, but never as entirely free as in text editor. In other words, the approach allows the arbitrary placement of Triples, but the precise representation (new lines, indentations, ...) is somewhat restricted. By editing data structures represented as text instead of text, we can provide advantages over textual editors, such as changing the IRI of a resource in all views of that resource in a document.[7]

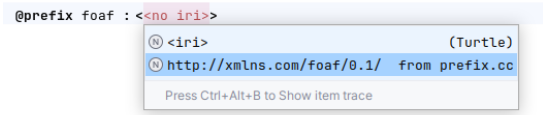
MPS provides completion of concepts thanks to the metamodel defined in the language structure[8]. These completion capabilities are extended with specific IRI completion features. In PrefixId, IRI completion is achieved through integration with prefix.cc, as illustrated in Figure 2b. For PrefixedName, once the prefix is known, completion for the name, as shown in Figure 2a, is provided by downloading the prefix namespace, assuming it contains a vocabulary or ontology, and processing it locally.

[7] One can argue that renaming references is possible with traditional textual IDEs, but in such approaches, the text is first parsed to an AST to find the scope of references, while we only have the AST that we immediately manipulate.

[8] Code completion in MPS is implemented within the editor aspect of concepts, through side transforms and node substitutions. While MPS provides default completions based on language structure and constraints, developers can extend these with custom implementations. An alternative approach involves grammarcells[21], which parse small text fragments into AST nodes.



(a) PrefixName completion for foaf prefix.



(b) PrefixId completion with prefix.cc suggestion for foaf.

Figure 2: Completion for some Turtle concepts.

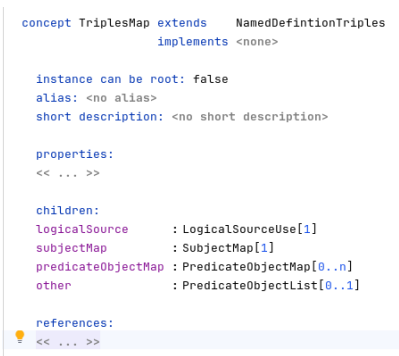
When importing a textual Turtle file, traditional Turtle parsers ignore empty lines and comments as their goal is to construct an RDF Graph. However, our parser retains them when at the top position to maintain visual resemblance with the original imported file. Comments in other positions are lost, it is beyond the scope of our prototype to support comments at all places in the projectional editor, as we need to reserve special slots for them in the AST and editor.

Converting the Turtle AST into text is straightforward as the DSL was build bottom-up from the grammar. An RDF Graph can also be constructed out of the Turtle AST, in which case during the conversion, the triples are annotated with the AST node responsible for their generation. One usage of this RDG Graph is validating the document against SHACL shapes, highlighting the invalid constructs thanks to the origin information. Another use case is to use the RDF Graph of the Turtle document to construct RML concepts out of it.

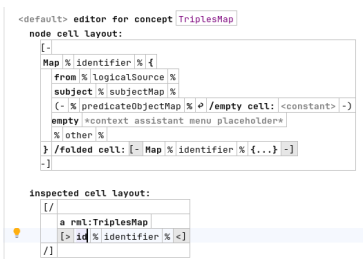
4.2. RML in GRAPE

RML is implemented as an extension of the previously introduced Turtle language. The RML structures are modelled as an interpretation of the specifications and shapes, leading to bottom-up DSLs that stay at the same level of abstraction as the specifications.

As a formal interpretation of the specifications, modifications have been made to fit the object-oriented model of structures and also to favor the simplicity of the AST. For AST simplicity, support for constant and references shortcut property has been removed. Instead, the editor assists in entering commonly used constructions by prefilling them. Also, it makes it easier for users to switch between a constant and other types of expression. Turtle concepts can be again used inside RML concepts to extend them, for example, in the TriplesMap concept Figure 3.



(a) TriplesMap structure. The other child contains Turtle nodes.



(b) TriplesMap editor takes distance with the Turtle syntax to provide a view organized with indentation and braces.

(c) TriplesMap RML-to-Turtle generator

Figure 3: TriplesMap in the RML(-Core) language contains an identifier (via inheritance), a logical source, a subject map, and a list of predicate object maps, but can also contain turtle predicate objects that will be attached to the subject of the triples map. It also inherits of the turtle TriplesStatment and therfore can be used instead of standard subjectTriples for example.

The notation used in the RML editor differs from its Turtle counterpart, aligning more closely with a block-based programming language. This design choice is used in all RML editors to maintain internal coherence and to take distance with the turtle syntax.

A notable example is the `rml:PredicateObjectMap`, which allows multiple predicates to be associated with multiple objects. In GRAPE, this structure is represented using braces { }, clarifying the relationships by grouping predicates and objects side by side in a mathematical notation.

Given the challenges encountered when authoring function executions in RML, an alternative notation has been proposed for FunctionExecution: **`function`** *`functionName`*`(`*`paramName`*` = `*`paramValue`*`)`. This notation resembles function calls in programming languages, `functionName(paramName: paramValue)`, and is similar to YARRRML one-line function syntax.

GRAPE permits editing existing RML mappings by importing exiting turtle files with RML mappings. The file is first imported to a Turtle AST, then the RDF Graph is constructed out of it. This graph is then used to construct high-level RML structures while leveraging origin information to retain relevant Turtle elements where appropriate. The steps of this transformation process are detailed in Figure 4. This approach of creating higher level abstraction is analogous to Yatter[8] for YARRRML.

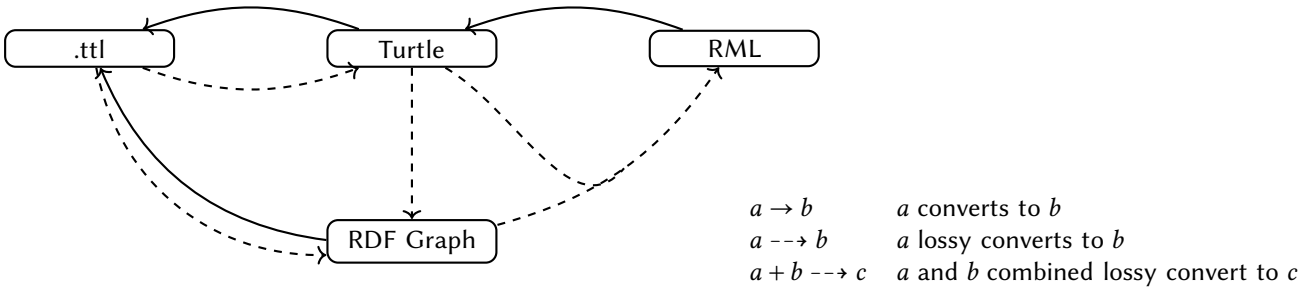


Figure 4: The transformation processes between Turtle files, the Turtle DSL, the RML DSL, and RDF Graph.

To complete the IDE integration, GRAPE has a basic integration of a RML engin, BURP [22], to be able to run the mappings from within the tool. Launching the mapping can be done visually, and the user can choose the working directory, optional base IRI and output file. The console output of the tools is visible in the IDE.

5. Demonstration

To validate the feasibility of our approach, we provide an illustrative demonstration, a method for initial evaluation of early-stage ideas and prototypes[23].

We reuse the examples provided in the Matey publication [11] to facilitate the comparison with prior work. The examples related to persons and movies are illustrated through *screenshots* of the mapping documents in Section 5. *Videos* of the authoring process are also available at https://gitlab.uliege.be/JakubDuchateau/grape.

The execution of a mapping file, facilitated by the BURP integration, is presented in Figure 5, which displays the full application screen. To demonstrate our notation for function execution, Figure 7 illustrates the extraction of a substring, specifically excluding the first three characters. The successful import of a mapping from the editor is shown in Figure 8. Furthermore, we have demonstrated the extensibility of our RML notations through the implementation of the RML-IO and RML-FNML, that can be enabled at runtime.

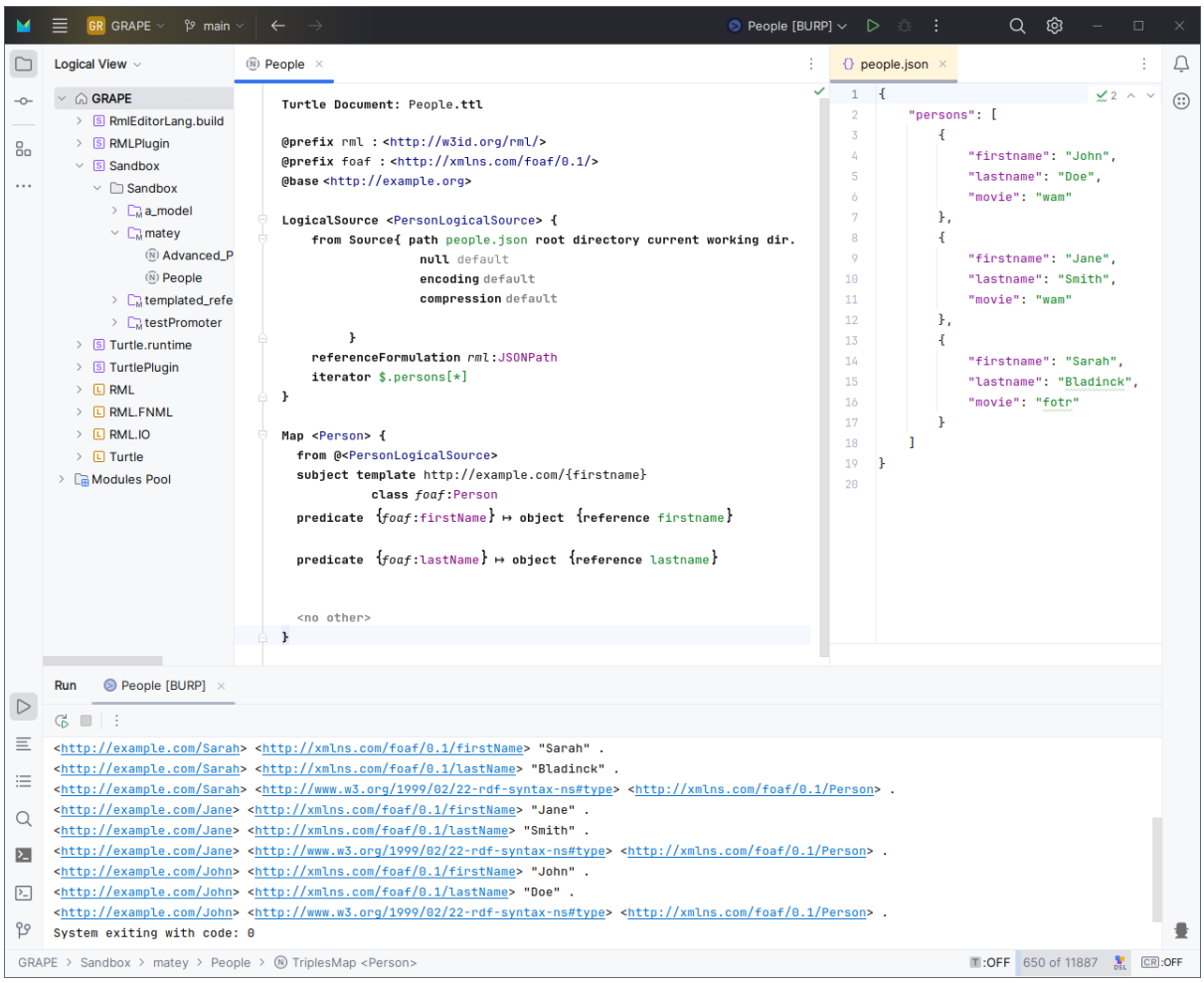


Figure 5: Screenshot showing the execution of the *people* mapping, with (left to right) the logical view, mapping document, JSON data, and the resulting RDF graph in the console at the bottom.

```
People    Advanced_PeopleAndMovies

Turtle Document: Advanced_PeopleAndMovies.ttl

@prefix ex : <http://example.org/>
@prefix burp : <http://w3id.org/burp/>
@prefix rml : <http://w3id.org/rml/>
@prefix foaf : <http://xmlns.com/foaf/0.1/>
@prefix schema : <http://schema.org/>
@base <http://example.com/>

Map <Person> {
  from LogicalSource{
    from Source{ path person.json root directory current working dir.
                 null default
                 encoding default
                 compression default

        }
    referenceFormulation rml:JSONPath
    iterator $.person[*]
}
  subject template person/${firstname}
           class foaf:Person
  predicate {ex:name} ↦ object {reference firstname}

  predicate {ex:likes} ↦ object { parentTriplesMap <Movie>
                                  join on child reference movie
                                          parent reference slug }

  <no other>
}

Map <Movie> {
  from LogicalSource{
    from Source{ path movies.csv root directory default
                 null default
                 encoding default
                 compression default

        }
    referenceFormulation rml:CSV
    iterator <row>
}
  subject template movie/${slug}
           class schema:Movie
  predicate {schema:name} ↦ object {reference title}

  predicate {ex:year} ↦ object {reference year}

  <no other>
}
```

Figure 6: People and Movie Mapping Example

```
Map <Person> {
  from @<PersonLogicalSource>
  subject template http://example.com/{firstname}
           class foaf:Person
  predicate {foaf:firstName} ↦ object {reference firstname}

  predicate {foaf:lastName} ↦ object { function grel:string_substring(
                                       grel:valueParam input value = reference lastname,
                                       grel:p_int_i_from from = 3
                                       ) }
```

Figure 7: Example of a substring function (highlighted in yellow).

```
Turtle Document: people.rml.ttl
Reload RDF Graph
@prefix rml : <http://w3id.org/rml/>
@prefix foaf : <http://xmlns.com/foaf/0.1/>
@base <http://example.org>

#  Example adapted from Matey
#  https://rml.io/yarrrml/matey/

<PersonLogicalSource>
  rml:source [
               rml:root rml:CurrentWorkingDirectory;
               rml:path "people.json" <lang or type>
             ];
  rml:referenceFormulation rml:JSONPath;
  rml:iterator "$.persons[*]" <lang or type>
  .

<Person>
  rml:logicalSource <PersonLogicalSource>;
  rml:subjectMap [
                   rml:template "http://example.com/{firstname}" <lang or type>;
                   rml:class foaf:Person
                 ];
  rml:predicateObjectMap [
                           rml:predicate foaf:firstName;
                           rml:objectMap [ rml:reference "firstname" <lang or type> ]
                         ];
  rml:predicateObjectMap [
                           rml:predicate foaf:lastName;
                           rml:objectMap [ rml:reference "lastname" <lang or type> ]
                         ]
  .
```

(a) Person Mapping imported in Turtle

```
Turtle Document: people.rml.ttl

@prefix rml : <http://w3id.org/rml/>
@prefix foaf : <http://xmlns.com/foaf/0.1/>
@base <http://example.org>

#  Example adapted from Matey
#  https://rml.io/yarrrml/matey/

LogicalSource <PersonLogicalSource> {
    from Source{ path people.json root directory current working dir.
                 null default
                 encoding default
                 compression default

        }
    referenceFormulation rml:JSONPath
    iterator $.persons[*]
}
Map <Person> {
  from @<PersonLogicalSource>
  subject template http://example.com/{firstname}
           class foaf:Person
  predicate {foaf:firstName} ↦ object {reference firstname}

  predicate {foaf:lastName} ↦ object {reference lastname}

  <no other>
}
```

(b) The turtle document promoted to RML structures

Figure 8: Import of a the Person Turtle Mapping

6. Conclusion and Future work

The authoring of KG construction mappings is challenges, and various approaches (languages, editors,...) have been proposed. We observed that approaches that provide lots of guidance tend to not provide much freedom to author (arbitrary RDF, or rigid editing order). We aimed to address this problem by proposing an editor that simulates a text editor but has built-in guidance to author well-formed Turtle and RML. In this work, we proposed GRAPE an IDE for RML build with a language-oriented programming approach. GRAPE provides guidance to the users while maintaining higher freedom than existing visual RML, thereby reducing the limitations users might experience with other tools.

GRAPE retains the flexibility of RDF vocabulary combination by enabling the representation of any RDF Node mixed with RML abstractions. Following the RML modularisation, we developed DSLs for RML-core, RML-IO sources, and RML-FNML function execution, closely following the specifications, ensuring users work with familiar abstractions. These abstractions, each with their own notations, integrate seamlessly into a single document, generating compliant RML mapping documents in textual Turtle, which can be then processed by RML engines. The modularity of this approach facilitates the integration of further abstractions, their semantics defined through model-to-model transformations.

The editor leverages projectional editing, implemented with JetBrains MPS, to unambiguously mix textual representation and integrated IDE features, such as run capabilities. Projectional editing of the document provides syntax error avoidance and guides the users with completions along the way to increase recognition instead of having to recall the specifications. The tools support importing existing Turtle files and reconstruct the GRAPE's RML abstractions from the turtle document.

Contributions

The contributions of this work include the development of a flexible and extensible editor that ensures syntactic correctness, supports multiple notations, and integrates features such as autocompletion and mapping execution. By leveraging a framework for creating Domain-Specific Languages, we introduce a novel approach for RML that allows the embedding of abstractions within a single document, enabling RML to be seamlessly integrated within Turtle and vice versa. Through examples and demonstrations, we present an alternative method for authoring RML documents, which may be more accessible and intuitive for users.

Limitations and Future work

While this work primarily demonstrates the capabilities of GRAPE through examples, a user evaluation is planned to investigate the tension field between guidance and freedom offered by projectional editing, and how it is perceived by the users.

We are waiting for the RML specifications to stabilise before proceeding with the implementation of the remaining RML features. While implementing further RML modules, we will probably need to improve extensibility or RML-Core. Furthermore, we aim to enhance GRAPE's integration of `referenceFormulation`, used in reference and templates query.

Acknowledgments

This work was supported by the Fonds de la Recherche Scientifique – FNRS under Grant No MIS F.4016.24.

Declaration on Generative AI

During the preparation of this work, the author(s) used GPT-4o and Grammarly to improve grammar and spelling check, paraphrase and reword, and translate words between French and English. After

using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] A. Iglesias-Molina, D. Van Assche, J. Arenas-Guerrero, B. De Meester, C. Debruyne, S. Jozashoori, P. Maria, F. Michel, D. Chaves-Fraga, A. Dimou, The RML Ontology: A Community-Driven Modular Redesign After a Decade of Experience in Mapping Heterogeneous Data to RDF, in: T. R. Payne, V. Presutti, G. Qi, M. Poveda-Villalón, G. Stoilos, L. Hollink, Z. Kaoudi, G. Cheng, J. Li (Eds.), The Semantic Web – ISWC 2023, Springer Nature Switzerland, Cham, 2023, pp. 152–175. doi:10.1007/978-3-031-47243-5_9.

[2] B. Myers, S. E. Hudson, R. Pausch, Past, present, and future of user interface software tools, ACM Trans. Comput.-Hum. Interact. 7 (2000) 3–28. doi:10.1145/344949.344959.

[3] A. C. Kay, The early history of Smalltalk, Association for Computing Machinery, New York, NY, USA, 1996, p. 511–598. URL: https://doi.org/10.1145/234286.1057828.

[4] K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, J. Zhao, Prov-o: The prov ontology, 2013. URL: https://www.w3.org/TR/prov-o/.

[5] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, The state of the art in language workbenches - conclusions from the language workbench challenge, in: M. Erwig, R. F. Paige, E. V. Wyk (Eds.), Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings, volume 8225 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 197–217. doi:10.1007/978-3-319-02654-1_11.

[6] M. Voelter, B. Kolb, T. Szabó, D. Ratiu, A. Van Deursen, Lessons learned from developing mbeddr: A case study in language engineering with MPS, Software & Systems Modeling 18 (2019) 585–630. doi:10.1007/s10270-016-0575-4.

[7] D. Van Assche, T. Delva, G. Haesendonck, P. Heyvaert, B. De Meester, A. Dimou, Declarative RDF graph generation from heterogeneous (semi-)structured data: A systematic literature review, Journal of Web Semantics 75 (2023) 100753. doi:10.1016/j.websem.2022.100753.

[8] A. Iglesias-Molina, D. Chaves-Fraga, I. Dasoulas, A. Dimou, Human-Friendly RDF Graph Construction: Which One Do You Chose?, in: I. Garrigós, J. M. Murillo Rodríguez, M. Wimmer (Eds.), Web Engineering, volume 13893, Springer Nature Switzerland, 2023, pp. 262–277. doi:10.1007/978-3-031-34444-2_19.

[9] P. Heyvaert, A. Dimou, A.-L. Herregodts, R. Verborgh, D. Schuurman, E. Mannens, R. Van De Walle, RMLEditor: A Graph-Based Mapping Editor for Linked Data Mappings, in: H. Sack, E. Blomqvist, M. d'Aquin, C. Ghidini, S. P. Ponzetto, C. Lange (Eds.), The Semantic Web. Latest Advances and New Domains, volume 9678, Springer International Publishing, Cham, 2016, pp. 709–723. doi:10.1007/978-3-319-34129-3_43.

[10] A. Crotti Junior, C. Debruyne, D. O'Sullivan, Juma: An Editor that Uses a Block Metaphor to Facilitate the Creation and Editing of R2RML Mappings, in: E. Blomqvist, K. Hose, H. Paulheim, A. Ławrynowicz, F. Ciravegna, O. Hartig (Eds.), The Semantic Web: ESWC 2017 Satellite Events, Springer International Publishing, Cham, 2017, pp. 87–92. doi:10.1007/978-3-319-70407-4_17.

[11] P. Heyvaert, B. De Meester, A. Dimou, R. Verborgh, Declarative Rules for Linked Data Generation at Your Fingertips!, in: A. Gangemi, A. L. Gentile, A. G. Nuzzolese, S. Rudolph, M. Maleshkova, H. Paulheim, J. Z. Pan, M. Alam (Eds.), The Semantic Web: ESWC 2018 Satellite Events, Springer International Publishing, Cham, 2018, pp. 213–217. doi:10.1007/978-3-319-98192-5_40.

[12] P. R. Aryan, F. J. Ekaputra, E. Kiesling, A. M. Tjoa, K. Kurniawan, RMLx: Mapping interface for integrating open data with linked data exploration environment, in: 2017 1st International Conference on Informatics and Computational Sciences (ICICoS), 2017, pp. 113–118. doi:10.1109/ICICOS.2017.8276347.

[13] C. A. Knoblock, P. Szekely, J. L. Ambite, A. Goel, S. Gupta, K. Lerman, M. Muslea, M. Taheriyan, P. Mallick, Semi-automatically Mapping Structured Sources into the Semantic Web, in: D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, E. Simperl, P. Cimiano, A. Polleres, O. Corcho, V. Presutti (Eds.), The Semantic Web: Research and Applications, volume 7295, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 375–390. doi:10.1007/978-3-642-30284-8_32.

[14] A. Iglesias-Molina, L. Pozo-Gilo, D. Dona, E. Ruckhaus, D. Chaves-Fraga, O. Corcho, Mapeathor: Simplifying the Specification of Declarative Rules for Knowledge Graph Construction, in: ISWC 2020 (Demos/Industry), volume 2721, CEUR, 2020. URL: https://ceur-ws.org/Vol-2721/paper488.pdf.

[15] A. Van Deursen, P. Klint, J. Visser, Domain-specific languages: an annotated bibliography, ACM SIGPLAN Notices 35 (2006) 26–36. doi:10.1145/352029.352035.

[16] M. Voelter, Language and IDE modularization and composition with MPS, in: R. Lämmel, J. Saraiva, J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers, Springer, 2013, pp. 383–430. doi:10.1007/978-3-642-35992-7_11.

[17] M. Voelter, J. Siegmund, T. Berger, B. Kolb, Towards User-Friendly Projectional Editors, in: B. Combemale, D. J. Pearce, O. Barais, J. J. Vinju (Eds.), Software Language Engineering, Springer International Publishing, Cham, 2014, pp. 41–61. doi:10.1007/978-3-319-11245-9_3.

[18] M. Fowler, Language Workbenches: The Killer-App for Domain Specific Languages?, 2005. URL: https://martinfowler.com/articles/languageWorkbench.html.

[19] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, G. Carothers, Rdf 1.1 turtle: Terse rdf triple language, W3C Recommendation, 2014. URL: https://www.w3.org/TR/2014/REC-turtle-20140225/, latest published version: https://www.w3.org/TR/rdf12-turtle/.

[20] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, G. Carothers, G. Kellogg, D. Tomaszuk, Rdf 1.2 turtle: Terse rdf triple language, W3C Working Draft, 2025. URL: https://www.w3.org/TR/2025/WD-rdf12-turtle-20250109/, latest published version: https://www.w3.org/TR/rdf12-turtle/.

[21] M. Voelter, T. Szabó, S. Lisson, B. Kolb, S. Erdweg, T. Berger, Efficient development of consistent projectional editors using grammar cells, in: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016, Association for Computing Machinery, New York, NY, USA, 2016, pp. 28–40. doi:10.1145/2997364.2997365.

[22] D. Van Assche, C. Debruyne, BURPing through RML test cases, in: KGCW 2024 : Knowledge Graph Construction 2024 : Proceedings of the 5th International Workshop on Knowledge Graph Construction Co-Located with 21th Extended Semantic Web Conference (ESWC 2024), volume 3718, CEUR, 2024. URL: https://ceur-ws.org/Vol-3718/paper4.pdf.

[23] K. Peffers, T. Tuunanen, M. A. Rothenberger, S. Chatterjee, A Design Science Research Methodology for Information Systems Research, Journal of Management Information Systems 24 (2007) 45–77. doi:10.2753/MIS0742-1222240302.