Shacled Turtle: Schema-Based Turtle Auto-Completion

Julian Bruyat, Pierre-Antoine Champin, Lionel Médini, Frederique Laforest

Shacled Turtle: Schema-Based Turtle Auto-Completion

Julian Bruyat[1], Pierre-Antoine Champin[1,2,3], Lionel Médini[1] and Frédérique Laforest[1]

[1]Univ Lyon, INSA Lyon, CNRS, UCBL, LIRIS, UMR5205, F-69621 Villeurbanne, France
[2]W3C, Sophia Antipolis, France
[3]University Côte d'Azur, Inria, CNRS, I3S (UMR 7271), France

Abstract

Producing RDF documents has always been a tedious task. To make it easier, most approaches propose an abstraction like forms to produce the data. In this paper, we propose Shacled Turtle, a method and a tool to ease the editing of RDF documents with auto-completion, based on RDFS and/or SHACL schemas. We also describe an experiment with volunteers to evaluate the usefulness of the tool, and we discuss its results.

Keywords

RDF, Auto-completion, SHACL

1. Introduction

Producing RDF data is a widely studied problem. In real life, most RDF datasets are either generated by tools to convert other formats to RDF or from data filled in a form by the user. However, the lack of good RDF editors is identified by the EasierRDF group as one of the many issues that can refrain developers from using RDF[1]. Indeed, many usages require to write relatively small pieces of RDF by hand: teaching the basics of RDF, describing ontologies, building graph patterns in SPARQL queries, declaring R2RML mappings [1]…Meta-ontologies, such as RDF Schema [2] (RDFS) or the Web Ontology Language [3] (OWL), have always played a central role in the RDF ecosystem. Recently, the necessity to validate RDF data appeared, giving birth to SHACL [4] and ShEx[2] to check the validity of graphs. But to the best of our knowledge, the literature has barely explored the ability to use inferential schemas or validating schemas to help users explicitly write RDF graphs.

Our research hypothesis in this work is that schemas can be used to provide useful suggestions to users when writing an RDF document. We developed Shacled Turtle[3] [5], a tool that uses RDFS and SHACL schemas to provide auto-completion for writing Turtle documents [6]. The

VOILA! 2022: 7th International Workshop on Visualization and Interaction for Ontologies and Linked Data, October 23, 2022, Virtual Workshop, Hangzhou, China, Co-located with ISWC 2022.

julian.bruyat@liris.cnrs.fr (J. Bruyat); pierre-antoine@w3.org (P. Champin); lionel.medini@liris.cnrs.fr (L. Médini); frederique.laforest@liris.cnrs.fr (F. Laforest)

CEUR Workshop Proceedings (CEUR-WS.org)

[1]https://github.com/w3c/EasierRDF/issues/35
[2]https://shex.io
[3]https://github.com/BruJu/shacled-turtle

purpose of this work is not the validation of documents, for which tools already exist, but only the assistance to the users in writing them.

The rest of this paper is structured as follows. In Section 2, we review the different available tools to produce RDF data. In Section 3 we present Shacled Turtle through a concrete example to give an intuition about the tool. In Section 4 we describe the architecture used to provide suggestions based on the chosen schema graph that is then detailed in Sections 5 and 6. In Section 7 we evaluate Shacled Turtle style of suggestions against the kind of suggestions provided by other similar tools from a user perspective. In Section 8 we discuss our contribution, the results and the experimental protocol. Finally, in Section 9 we conclude with the possible improvements.

2. Where do RDF Triples come from?

To produce RDF data, most approaches use some kind of abstraction, convert existing data or directly write programs that output RDF data.

2.1. RDF data production

The most popular abstractions are forms. Protégé [7] requires the user to fill forms to build their ontology and then generate the corresponding RDF file. The SHACL specification explicitly mentions the possibility to generate forms from property shapes, which has been implemented by systems like Schimatos [8]. Form generators have also been developed for the other shape language ShEx.
Converting non-RDF data is also a popular way to produce RDF data. Many tools have been developed: R2RML maps relational databases and tabular data to RDF by using mappings provided by the user, RML [9] extends the latter to support other kinds of data sources, RDF123 [10] aims to produce RDF data by using spreadsheets as an abstraction, JSON-LD [11] transforms JSON documents to RDF and is the way recommended by Google to add metadata to a website in order to improve its SEO (Search Engine Optimisation).

Even by using these approaches, users may still have to write RDF documents: the R2RML mappings must be described in RDF, users may want to fine tune the ontology produced by Protégé.

2.2. Current editors

Plugins for popular code editors have been developed, like *LNKD. tech Editor* [4] and *RDF and SPARQL* [5] for the JetBrain suite. A language server for Turtle has recently been developed by Stardog Union[6]. But all these plugins mainly focus on syntactic checking and coloration.

In [12], Rafes et al. list some of the expected features from a SPARQL auto-completion module. They identify 3 major categories: suggestion of snippets, prefix declaration and auto completion for Internationalized Resource Identifiers (IRIs). Snippets suggestion is described as being mostly

[4] https://plugins.jetbrains.com/plugin/12802-lnkd-tech-editor
[5] https://sharedvocabs.com/products/rdfandsparql/
[6] https://marketplace.visualstudio.com/items?itemName=stardog-union.vscode-langserver-turtle

requested by experienced users "and can be seen as the step after suggesting terms". Prefix auto-completion is deployed by most editors, through the use of the prefix.cc API[7].

To the best of our knowledge, IRI suggestions in all RDF document editors, like *RDF and SPARQL* and *Yasgui* [13] are limited to proposing all the terms that exists in a given ontology. *Yasgui* filters the list of suggestions depending on the position: for example if the current term is a predicate, all properties in the ontology are displayed and other terms are discarded. This approach is best suited for small ontologies, as for big ontologies, like schema.org[8], the number of suggestions can reach hundreds, making it impractical for users.

Some SPARQL editors like the Flint Sparql Editor[9] or the one presented by Gombos and Kiss in [14] uses intermediate SPARQL queries to help users write their queries. Sparqlis [15] also uses this approach but goes further by exposing an interface in natural language, removing the need for the end-user to know SPARQL. In [16], De la Parra and Hogan first compute the relationships between all types and predicates in the graph, and use the result of this computation to provide auto-completion when the user builds their SPARQL query. All these approaches resort on using the actual data to produce the effective schema of the graph. But in our case, as we are interested in writing new data, these kinds of approaches are not applicable. Hence instead of using the effective schema of the graph we will rely on the expected schema as specified by an RDFS ontology or a set of SHACL shapes.

3. Shacled Turtle usage example

Shacled Turtle is implemented as a Code Mirror 6[10] extension that provides support for the Turtle language.

The selling key-point of Shacled Turtle is the auto-completion module. While most advanced editors suggest all terms from the ontology, Shacled Turtle narrows the list to the parts of the ontology that are related to the currently edited resource. We consider that most resources in an RDF graph must be typed. When the type of a resource is known, it is likely that the predicates related to the known types will be used to describe it. Figure 1 shows a concrete usage example: we defined that `ex:alice` is a `s:Person` and then we start to write a new triple. The suggestion engine considers that we probably want to use a predicate related to persons, like `s:nationality` or `s:name`, and does not suggest terms like `s:numTracks` or `s:measurementTechnique` that are related to other types.

4. Shacled Turtle general architecture

Figure 2 shows the general architecture of Shacled Turtle.

0. Before the interaction starts, a preprocessing phase is performed. The content of a *schema graph* is converted into *inference rules* and *suggestion rules* by the *schema to rules converter*. This schema can be written either in RDFS, in SHACL or a mix of both.

[7] https://prefix.cc
[8] https://www.schema.org
[9] http://fr.dbpedia.org/sparqlEditor
[10] https://codemirror.net/6/

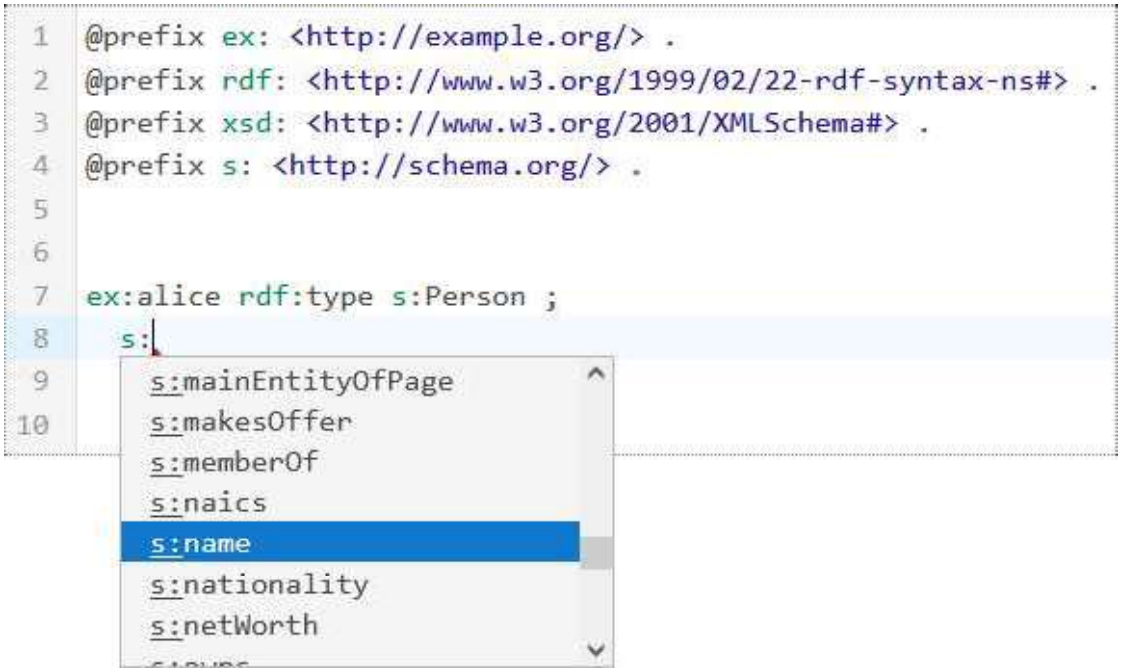


Figure 1: For a subject of type Person, Shacled Turtle only suggests predicates related to this type in the Schema ontology

1. When the user is writing an RDF graph, the *inference engine* uses all *complete triples* to deduce the types of all resources and the list of shapes that they should comply with. These results are stored in the *meta graph*.
2. When the user is writing a new *incomplete triple*, after the subject has been written, *i.e.* on writing the predicate or on writing the object, the *suggestion engine* queries the *meta graph* for the list of all the types and shapes of the subject. It will then return to the user:
   - If the incomplete triple only has a subject, the list of all predicates related to the types and shapes of the subject.
   - If the incomplete triple has a subject and a predicate, a list of resources depending on the types and shapes of the subject and the predicate[11].

We will first describe the basics of all the components used by the *interaction loop* in Section 5, in particular describe how the rule systems used by the *inference engine* and the *suggestion engine* work. Then we will describe how the *preprocessing* translates the schema graph into *inference and suggestions rules* in Section 6.

5. The interaction loop

The **interaction loop** comprises all operations performed when the user uses the editor.

[11]Note that for some predicates like `rdf:type`, the suggestion may be independent of the list of types and shapes of the subject.

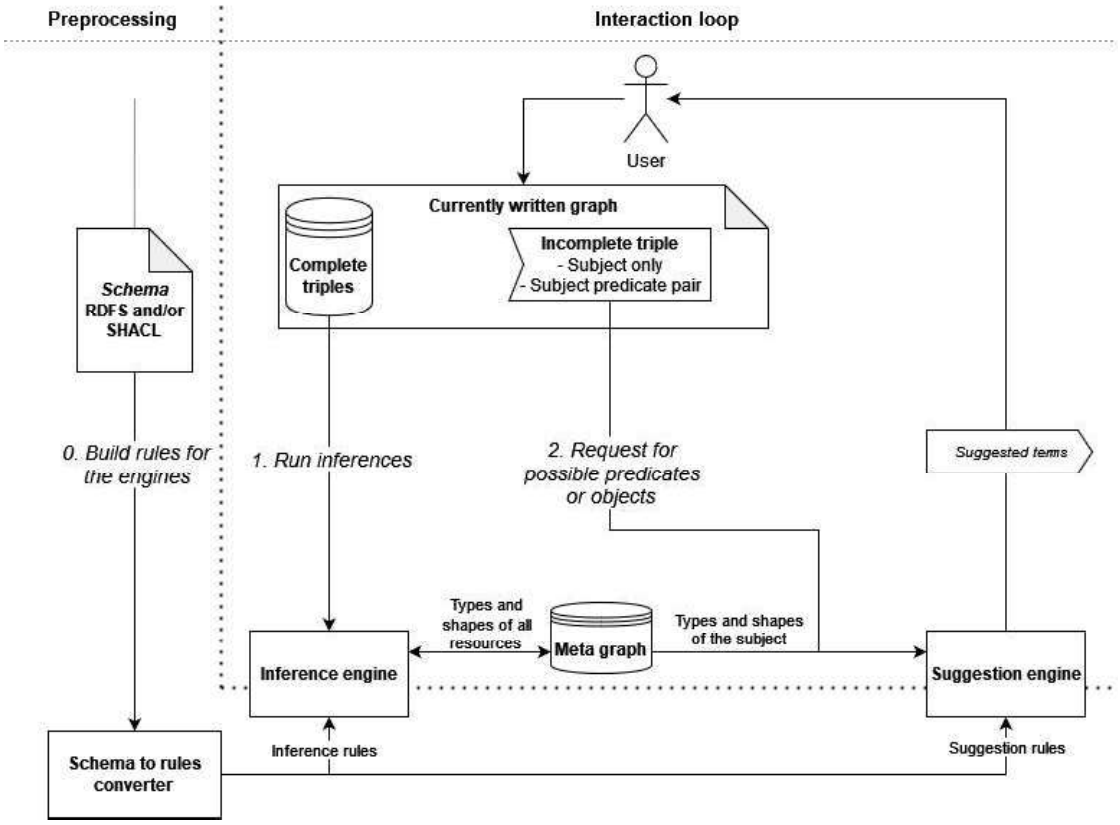


Figure 2: The different components of Shacled Turtle

5.1. The graphs

During the interaction loops, two different graphs are used:

- The *currently written graph* is the graph in the text editor. It is composed of two different parts:
  - The *completed triples*, triples for which the subject, the predicate and the object are known. These triples are used to power up the *inference engine* and produce triples for the *meta graph*.
  - The *incomplete triple* that the user is currently editing. If the subject of this incomplete triple is known, the *suggestion engine* and the *meta graph* will be requested to provide a list of *suggested terms*. If other triples are incomplete, they are ignored by the engine.
- The *meta graph* is the graph that stores triples produced by the *inference engine*. Its role is to store, for each resource, the list of all known types, and the list of shapes that the resource should comply with. The content will then be used by the *suggestion engine*, in particular to know the list of types and shapes of the subject of the *incomplete triple*.

5.2. The inference engine

We want our system to be able to deduce both the deducible types from an RDFS ontology and to be able to list the shapes a resource must comply with.

These inferences are specified by *inference rules* (see Table 1 and 3). These rules go beyond the ones traditionally used for RDFS, but do not need to capture the full semantics of SHACL as we are not aiming at validating the graph.

Each inference rule has the form:

$$\frac{DataTriple? \quad SourceMetaTriple?}{ProducedMetaTriple}$$

where:

- The body can require
  - $DataTriple?$: 0 or 1 complete triple from the *currently written graph*.
  - $SourceMetaTriple?$: 0 or 1 triple from the *meta graph*.
- The head must be a triple $ProducedMetaTriple$ that will be stored in the *meta graph* and its predicate must either be `rdf:type` or `:pathsOf`[12].

5.3. The suggestion engine

In the same way, the system relies on a set of rules to deduce the possible suggestions at run-time, from the meta graph and the incomplete triple.

We suppose that $?s, ?p, ?o$ are variables and A and P are IRIs.

Each suggestion rule has the form:

$$\frac{IncompleteTriple \quad MetaTripleCondition?}{Suggestion}$$

where:

- The $IncompleteTriple$ is either
  - $(?s, \dots, \dots)$ for applying the rule when only the subject of the *incomplete triple* is known.
  - $(?s, A, \dots)$ for applying the rule when both the subject and predicate are known.
- The $MetaTripleCondition?$ is optional, and either
  - A triple pattern of the form $(?s, P, ?o)$ that is searched in the *meta graph*, where $?s$ is the subject occurring in $IncompleteTriple$.
  - "No info on $?s$", for applying the rule only if there are no types or shapes known for the resource `?s`.
  - $none$ when no condition holds on the meta graph content.
- The $Suggestion$ is either
  - $suggest(A)$ to add A to the list of suggested terms.
  - $suggestAll(?p, ?o)$ to add to the list of suggested terms all resources α such that $(\alpha, ?p, ?o)$ is in the *meta graph*.

[12]The triple $(u, :pathsOf, s)$ means that for the resource u we should suggest the paths specified by the shape s.

Table 1
Transformation of triples in the schema graph into inference and suggestion rules.

Triple in schema graph	***Inference rules***	***Suggestion rules***
	$\frac{(?u, rdf{:}type, ?t) \quad none}{(?u, rdf{:}type, ?t)}$ $\frac{none \quad (?u, rdf{:}type, ?t)}{(?t, rdf{:}type, rdfs{:}Class)}$	$\frac{(?u, \ldots, \ldots) \quad \text{No info on } ?u}{suggest(rdf{:}type)}$ $\frac{(?u, rdf{:}type, \ldots) \quad none}{suggestAll(rdf{:}type, rdfs{:}Class)}$
$(P', rdfs{:}domain, T)$ $\forall P = P'$ or P' subproperty of P	$\frac{(?u, P, ?v) \quad none}{(?u, rdf{:}type, T)}$	$\frac{(?u, \ldots, \ldots) \quad \text{No info on } ?u}{\text{suggest(P)}}$ $\frac{(?u, \ldots, \ldots) \quad (?u, rdf{:}type, T)}{suggest(P)}$
$(P', rdfs{:}range, T)$ $\forall P = P'$ or P' subproperty of P	$\frac{(?u, P, ?v) \quad none}{(?v, rdf{:}type, T)}$	$\frac{(?u, P, \ldots) \quad none}{suggestAll(rdf{:}type, T)}$
$(P, s{:}domainIncludes, T)$		$\frac{(?u, \ldots, \ldots) \quad (?u, rdf{:}type, T)}{suggest(P)}$
$(P, s{:}rangeIncludes, T)$		$\frac{(?u, P, \ldots) \quad none}{suggestAll(rdf{:}type, T)}$
$(S, rdf{:}type, sh{:}NodeShape)$	$\frac{none \quad (?u, rdf{:}type, S)}{(?u, :pathsOf, S)}$	
$(S, sh{:}targetNode, U)$	$\frac{none \quad none}{(U, :pathsOf, S)}$	
$(S, sh{:}targetClass, T)$	$\frac{none \quad (?u, rdf{:}type, T)}{(?u, :pathsOf, S)}$	$\frac{(?u, rdf{:}type, \ldots) \quad none}{suggest(T)}$
$(S, sh{:}subjectsOf, P)$	$\frac{(?u, P, ?v) \quad none}{(?u, :pathsOf, S)}$	$\frac{(?u, \ldots, \ldots) \quad \text{No info on } ?u}{\text{suggest(P)}}$ $\frac{(?u, \ldots, \ldots) \quad (?u, :pathsOf, S)}{suggest(P)}$ $\frac{(?u, \ldots, \ldots) \quad none}{suggest(P)}$
$(S, sh{:}objectsOf, P)$	$\frac{(?u, P, ?v) \quad none}{(?v, :pathsOf, S)}$	$\frac{(?u, P, \ldots) \quad none}{suggestAll(:pathsOf, \text{S})}$
$(S1, sh{:}node, S2)$ and $S1$ is a node shape	$\frac{none \quad (?u, :pathsOf, S1)}{(?u, :pathsOf, S2)}$	

6. The preprocessing

We now describe the *preprocessing*, which is the step where the *schema to rules converter* converts the *schema* graph into *inference* and *suggestion rules* for the eponymous engines. It uses two kinds of transformations: rules that are built by searching all triples with a certain pattern in

the schema graph, and SHACL paths whose recursive nature will be handled by using finite state automata.

6.1. Rules built by looking up some triples pattern

Table 1 exposes the list of inference and suggestion rules that are generated from the schema graph. Note that the purpose of this tool is neither to infer all possible suggestions, nor to validate the graph, but to make suggestions that are as relevant as possible. This is a subjective criterion, as having either too few or too many suggestions would make the tool less useful. We will discuss this further in Section 8.

6.2. Rules built from SHACL Paths

Similarly to SPARQL paths, a SHACL path can be either a predicate path (an out-coming triple with a given predicate) or a composition of other paths with one of the following operators: inverse, sequence, alternative, repetition, kleene, and optional.

One issue with paths is that we want to be able to process complex paths, and provide suggestions at any point in the path. For example, for the sequence path `( :a :b )`, `:b` should be a suggested predicate for nodes targeted by `:a`.

Unit paths and virtual shapes Our solution is to split composite paths into what we consider unit paths. Unit paths are either predicate paths, *e.g.* `:owns`, or inverse paths of a predicate path, *e.g.* `[ sh:inversePath :ownedBy ]`. These unit paths are connected with *virtual shapes*, shapes that do not explicitly exist in the composite shape graph. The chain of all the unit paths through the virtual shapes is equivalent to the original composite path for the purpose of our suggestion engine.

Let us consider the shape graph on Listing 1. This shape graph means any node `?postalAddress` extracted from the SPARQL request on Listing 2 must comply with the node shape `s:PostalAddress`. For our purpose, it is equivalent to the shape graph on Listing 3 where we introduced a new shape, `ex:VirtualShape` that will act as the shape of all the matches for `?o` in the SPARQL request.

Overview on transforming SHACL Paths into rules. To process SHACL paths, we assume that:

- We can decompose any path into unit paths connecting virtual shapes.
- Processing a chain of unit predicate paths is similar to processing a string with a regex. Hence we can use finite-state automaton (FSA) to recognize if a chain of triples is recognized by a path.
- The only difference between a predicate path and an inverse predicate path is whenever the subject or the object variable is bound to a known value.

Based on these assumptions, to parse the SHACL path into a list of inference and suggestion rules, we first transform the path into an FSA, then we transform the FSA into rules.

Listing 1: An example shape

```
s:Person rdf:type sh:NodeShape ;
  sh:targetClass s:Person ;
  sh:property [
    sh:path (
      s:worksFor
      s:address
    ) ;
    sh:node s:PostalAddress
  ] .
```

Listing 2: SPARQL query to get all the resources targeted by the property shape contained by `s:Person`

```
SELECT ?postalAddress WHERE {
  # All resources targeted by the shape
  ?person rdf:type s:Person .
  # Travel the path
  ?person s:worksFor ?o .
  ?o s:address ?postalAddress .
}
```

Listing 3: The same shape graph with a virtual shape

```
s:Person rdf:type sh:NodeShape ;
  sh:targetClass s:Person ;
  sh:property [
    sh:path s:worksFor ;
    sh:node ex:VirtualShape01
  ] .

ex:VirtualShape01
  rdf:type sh:NodeShape ;
  sh:property [
    sh:path s:address ;
    sh:node ex:PostalAddress
  ] .
```

From SHACL paths to FSA. We build the FSA that describes the path P by composition. Predicate paths produce an FSA with two states and only one transition. The FSA of other paths, that are composite paths, are built by combining the automaton of their components in some way. The transition symbol used by all the produced automaton are composed by combining either the sign $+$ for out-going edges or $-$ for incoming edges, with the predicate to travel. Table 2 describes all the composition rules, where we consider that p is any predicate, P, P1 and P2 are paths.

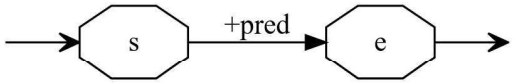
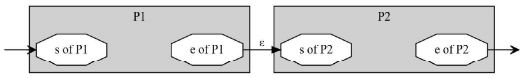
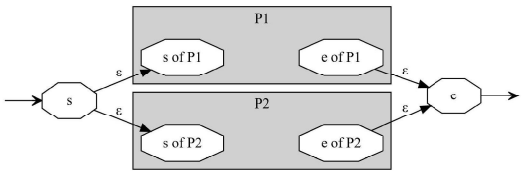
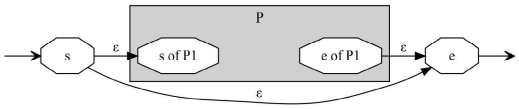
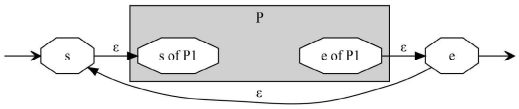
From FSA to rules. After minimization and determination, an FSA can be defined as one initial state, a set of final states and a set of transitions $(StartState, Symbol, EndState)$.

- We define m a total function from all states s of the FSA to RDF Nodes. For each state s, $m(s)$ is a fresh RDF node, *i.e.* it is not used elsewhere.
- The virtual shape mapped from the initial state of the FSA is a super-shape of the starting shape of the property shape
- If a destination shape is known for the property shape, it is declared as a sub-shape of all the final states of the FSA.
- Table 3 describes how to convert the transitions to inferences rules.

7. Evaluation

Shacled Turtle uses schemas to reduce the number of suggestions proposed to users, keeping only the most relevant ones. The underlying assumption is that this is more helpful for users

Table 2
Mapping from all kinds of path to automata

Kind and SHACL Syntax	***Regex equivalent***	***Built automaton***
Predicate `pred`	p	
Inverse `[ sh:inversePath P ]`	None	Take automaton P Inverse all transitions Transform all + into - Transform all - into +
Sequence `( P1 P2 )`	P1 P2	
Alternate `[ sh:alternatePath ( P1 P2 ) ]`	(P1 | P2)	
Zero or one `[ sh:zeroOrOnePath P ]`	P?	
One or more `[ sh:oneOrMorePath P ]`	P+	
Zero or more `[ sh:zeroOrMorePath P ]`	P*	Equivalent to (P+)?

than a less selective suggestion engine. In order to evaluate the validity of this assumption, we asked volunteers to translate two texts into Turtle documents by using a given ontology. The produced documents were expected to be constituted of approximately 10 triples. One of the documents had to be written by using our auto-completion engine, the other by using an auto-completion engine similar to the one used by YASGUI, *i.e.* that displays all the terms of the ontology. The order of the two different documents and of the two auto-completion engines was randomized.

We used two different schemas:

- The Schema.org ontology. For this session, we used the RDF schema graph published on Github by Schema.org[13]. We slightly altered the graph to transform the cases where a pred-

[13]https://github.com/schemaorg/schemaorg/blob/main/data/releases/14.0/schemaorg-all-https.ttl

Table 3
Converting the transitions of the produced FSA to rules

Transition	*Inference rules*	*Suggestion rules*
$(S, +P, E)$	$\frac{(?u, P, ?v) \qquad (?u, :pathsOf, m(S))}{(?v, :pathsOf, m(E))}$	$\frac{(?u, \dots, \dots) \qquad (?u, :pathsOf, m(S))}{suggest(P)}$ $\frac{(?u, P, \dots) \qquad (?u, :pathsOf, m(S))}{suggestAll(:pathsOf, m(E))}$
$(S, -P, E)$	$\frac{(?u, P, ?v) \qquad (?v, :pathsOf, m(S))}{(?u, :pathsOf, m(E))}$	

icate only had one value for `schema:domainIncludes` or `schema:rangeIncludes` to `rdfs:domain` and `rdfs:range` to help the *inference engine* of Shacled Turtle. This alteration has no impact on the naive suggestion engine. As said previously, Schema.org is a big ontology with thousands of terms. For this session, we had 23 volunteers, 21 of them were Semantic Web experts with more than 3 years of usage and 6 had already used the Schema.org ontology.

- *Friend of a friend* (foaf)[14]. As this ontology is defined by using mostly RDFS, it benefits fully from the *inference engine*. Moreover, it is a small ontology, with a few dozen of terms. For this session, we had 11 volunteers, 7 of them were Semantic Web experts and none declared to already have used the ontology.

After writing the two different RDF documents, one with Shacled Turtle and one without it, they were asked to grade on a Likert scale [17] their feeling about the usefulness of both completion engine (naive and Shacled Turtle) and if they preferred an auto-completion engine over another one. We also let users explain in a free field why they preferred one engine, if any; and another free field to collect general feedback. Finally, we measured how much time each volunteer took to write each document.

The whole evaluation was conducted online. We published the source code of the platform and the anonymized collected results on Github at https://github.com/BruJu/shacled-turtle-evaluation.

Of the 34 volunteers, 17 declared to have no preference towards an engine or the other. Six volunteers even admitted to have seen no difference between the two engines. The number of people that prefer one engine over another is almost equal for both engines.

When asked separately, all volunteers gave a similar rank to both engines, the worst case being a strong appreciation on an engine and a neutral appreciation on the other; but 21 users gave the same appreciation to both.

Using Shacled Turtle does not enable the user to complete the task faster: 20 volunteers were faster to complete the second task than the first, regardless of if Shacled Turtle is the first engine or the second, and 14 took about the same time.

[14] https://xmlns.com/foaf/spec/

8. Discussion

In this section, we study some of the most recurring comments made by the volunteers about the tool to have a better understanding of what can be improved in Shacled Turtle.

Relevance of suggestions. Five volunteers showed a high enthusiasm about the approach and their comments showed that they understood well the purpose of the tool. In particular, two of them appreciated that the tool leads to less errors, feeling more confident about the produced graph.

However, in Section 6.1, we mentioned that the choice of which suggestions to filter out is, to some extent, arbitrary, and could lead to false negatives.

The question arises especially in the case of SHACL shapes: we suggest only predicates that are mentioned in the shape(s) of the subject, but unless these shapes are flagged with `sh:closed true`, they actually do not disallow *other* predicates. Similar issues may apply with RDFS classes, because an instance of a class might still be an instance of another one.

Indeed, three volunteers complained about the fact that Shacled Turtle produced less suggestions than the other engine. 21 volunteers ranked both engines similarly, and six of them explicitly reported that they did not notice any difference, suggesting that there is no clear benefit in reducing the overall number of suggestions.

Other filtering strategy. Most auto-completion engines enable users to filter the list of suggestions by name. In Code Mirror, and therefore in Shacled Turtle, when the user types for example `s:na`, the system will only show the terms that contain the characters `s:na` in that order (e.g. `s:familyName` or `s:eventStatus`). A common practice to find a desired term is to opportunistically reduce the list of terms using the filtering by name. Then when the user considers the list of terms to be short enough, they look further at the displayed terms. The responses of the volunteers indicate that they proceeded that way.

Therefore it might be more valuable to *promote* the suggestions we deem relevant than to filter out the others, and leave it to the user reduce the number of suggestions using filtering by name. Once a suggestion list is filtered out by the user, we think that Shacled Turtle could provide an efficient strategy to help the user pick the right term, in conjunction with manual filtering by name.

The importance of good documentation. Shacled Turtle shows, with each suggested term, a description (`rdfs:comment`) of that term when provided by the schema. While the query GUI of Wikidata does the same, because Wikidata IRIs are opaque, many other suggestions engine do not. During our experiment, seven volunteers reported that the descriptions of the terms are important, as they complained when descriptions were missing or incomplete, either because of bugs during the early stages of the experiment or because of the used schema. Five volunteers reported to have consulted the ontology online documentation to check how to use the ontology and have a better idea of the usage of the terms and their links. At the opposite, a volunteer reported that thanks to this tool, they fortunately did not feel the necessity to consult the ontology documentation.

One of the volunteers explained that the domain and the range of a property can be more informative than a description. The *schema to rules converter* could also be used to enrich the descriptions to add the links between the predicates and the different types and shapes.

As mentioned previously, Shacled Turtle should not be used to filter out choices from contextual data, but to enrich the documentation. This could be changed by using Shacled Turtle to *promote* terms that we deem relevant, either by displaying them first in the list, by highlighting them, or both: it would solve the issue of users not seeing a difference. To increase the perceived reliability of the tool, the decision made should be explained to the users, *i.e.* in case of an incomplete triple with only a subject, displaying which type or shape of the subject is used to suggest each relevant predicate; and for an incomplete triple with only a missing object, which type or shape of the suggested objects is used to suggest them depending on the subject and the predicate.

9. Conclusion

In this paper, in order to tackle the problem of writing RDF documents by hand, we proposed Shacled Turtle, an auto-completion engine that resorts to a schema graph to suggest terms related to the types and shapes of the subject of the triple that the user is writing. The system relies on two different rule engines : an *inference engine* that deduces the list of types and shapes of all resources in the *currently written graph* and a *suggestion engine* that provides possible following terms. However, in our experiments, the users barely saw any difference between a naive approach, proposing all terms that are in the ontology, and our approach: to find appropriate terms, they preferred to rely on other strategies like filtering by name and reading the ontology online documentation. We explain this by the inability of our method to display explicit insights: the difference between Shacled Turtle and the naive approach is implicit, as it consists in showing less options.

As users are in quest of information, four aspects can be considered:

- Enriching the descriptions of the terms, both with information extracted from the *schema to rules converter* like the links between the predicates and the types and shapes, and with contextual information to explain why the system thinks a term may be relevant in the current *incomplete triple*.
- Instead of using Shacled Turtle to filter out irrelevant terms, promote these relevant terms in the list of all existing terms.
- Running an inference engine to provide the list of types of the resources when the user hovers the resource. While this is currently done for RDFS, it could be expanded to any inference rule-set like OWL.
- Using a SHACL validation report to report errors, *i.e.* as a linting tool. This would lead to more accurate information and more visible error.

Another perspective that is to propose snippets, i.e. complete set of triples, instead of simple paths. SHACL sequence paths are paths composed of other paths: instead of requiring the user to chain blank nodes for each path that composes the sequence path, a snippet could be suggested that would build all the intermediate blank nodes at once. This approach would better benefit from SHACL paths and offer a higher level of suggestion.

Acknowledgments

We would like to thank all the volunteers for their time and their very valuable feedback.

References

[1] S. Das, S. Sundara, R. Cyganiak, R2RML: RDB to RDF Mapping Language, W3C Recommendation, W3C, 2012. Https://www.w3.org/TR/2012/REC-r2rml-20120927/.
[2] R. Guha, D. Brickley, RDF Schema 1.1, W3C Recommendation, W3C, 2014. Https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.
[3] D. L. McGuinness, F. Van Harmelen, et al., Owl web ontology language overview, W3C recommendation 10 (2004) 2004.
[4] D. Kontokostas, H. Knublauch, Shapes Constraint Language (SHACL), W3C Recommendation, W3C, 2017. Https://www.w3.org/TR/2017/REC-shacl-20170720/.
[5] J. Bruyat, Bruju/shacled-turtle: Shacled turtle 0.0.3, 2022. URL: https://doi.org/10.5281/zenodo.6907388. doi:10.5281/zenodo.6907388.
[6] E. Prud'hommeaux, G. Carothers, RDF 1.1 Turtle, W3C Recommendation, W3C, 2014. Https://www.w3.org/TR/2014/REC-turtle-20140225/.
[7] M. A. Musen, The protégé project: a look back and a look forward, AI matters 1 (2015) 4–12.
[8] J. Wright, S. J. Rodríguez Méndez, A. Haller, K. Taylor, P. G. Omran, Schímatos: a shacl-based web-form generator for knowledge graph editing, in: International Semantic Web Conference, Springer, 2020, pp. 65–80.
[9] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, R. Van de Walle, Rml: a generic language for integrated rdf mappings of heterogeneous data, in: Ldow, 2014.
[10] L. Han, T. Finin, C. Parr, J. Sachs, A. Joshi, Rdf123: from spreadsheets to rdf, in: International Semantic Web Conference, Springer, 2008, pp. 451–466.
[11] P.-A. Champin, G. Kellogg, D. Longley, JSON-LD 1.1, W3C Recommendation, W3C, 2020. Https://www.w3.org/TR/2020/REC-json-ld11-20200716/.
[12] K. Rafes, S. Abiteboul, S. Cohen-Boulakia, B. Rance, Designing scientific sparql queries using autocompletion by snippets, in: 2018 IEEE 14th International Conference on e-Science (e-Science), IEEE, 2018, pp. 234–244.
[13] L. Rietveld, R. Hoekstra, Yasgui: not just another sparql client, in: Extended Semantic Web Conference, Springer, 2013, pp. 78–86.
[14] G. Gombos, A. Kiss, Sparql query writing with recommendations based on datasets, in: International Conference on Human Interface and the Management of Information, Springer, 2014, pp. 310–319.
[15] S. Ferré, Sparklis: An expressive query builder for sparql endpoints with guidance in natural language, Semantic Web 8 (2017) 405–418.
[16] G. de la Parra, A. Hogan, Fast approximate autocompletion for sparql query builders (2021).
[17] R. Likert, A technique for the measurement of attitudes., Archives of psychology (1932).