ABSTRACT

Developing and testing modern RDF-based applications often requires access to RDF datasets with certain characteristics. Unfortunately, it is very difficult to publicly find domain-specific knowledge graphs that conform to a particular set of characteristics. Hence, in this paper we propose RDFGraphGen, an open-source RDF graph generator that uses characteristics provided in the form of SHACL (Shapes Constraint Language) shapes to generate synthetic RDF graphs. RDFGraphGen is domain-agnostic, with configurable graph structure, value constraints, and distributions. It also comes with a number of predefined values for popular schema.org classes and properties, for more realistic graphs. Our results show that RDFGraphGen is scalable and can generate small, medium, and large RDF graphs in any domain.

1 Introduction

The acceptance of RDF [1] as a standard for knowledge representation, publication on the Web [3], and internally in organizations [7], has led to the development of many tools, libraries, and applications that work with or use RDF as a data model. During the development lifecycle of each of these, benchmarking and testing are important steps, which validate the usage scenarios and provide metrics about their real-world usability [4]. These benchmarks and test scenarios often require application-specific or domain-specific RDF datasets with certain characteristics, that are not always readily available. One possible solution to this is synthetic RDF datasets — datasets containing entities and data that are artificial but follow the structure, distribution, and vocabulary expected from applications or systems. They fulfill the need for data in a specific format, for a specific domain, application, or system to be tested or benchmarked [9, 10]. Depending on the specific needs, these synthetic RDF datasets can be small or consist of millions of different entities [8].

One of the more recent additions to the technology stack around RDF is the Shapes Constraint Language (SHACL) [2]. Its main purpose is defining structural constraints on RDF graphs, called *shapes*. SHACL is designed to validate RDF graphs to ensure their data quality. However, these shapes can also be used as *blueprints* for generating domain-agnostic RDF data. Recognizing this potential and aiming to address the need for an RDF data generator, this paper presents RDFGraphGen. RDFGraphgen is, to the best of our knowledge, the first *SHACL-based* RDF data generation tool. As opposed to the usual task of *validating* RDF data, we use SHACL shapes for *generating* RDF data.

Hence, in this paper we describe the design and implementation of RDFGraphGen — a domain-independent generator of RDF graphs based on SHACL shapes. SHACL shapes are interpreted as a description of the desired structure of an RDF graph, and in accordance with this view the corresponding RDF triples are generated. RDFGraphGen is domain-agnostic: the source SHACL shapes can be from any domain. The generator can generate an RDF graph with a specified scale factor that determines the number of generated entities, providing flexibility for end-users. RDF data generated by RDFGraphGen can be used for benchmarking, testing, quality control, or other similar tasks in the

application lifecycle of systems using RDF. Additionally, RDFGraphGen is open-source[1] and available as a ready-to-use Python package[2], under the MIT license.

This paper is organized as follows. Section 2 reviews related work and existing tools. Section 3 describes the design and the underlying algorithms of RDFGraphGen. Section 4 illustrates its features, usability, and data generation through a practical example. Section 5 outlines the results of the performance evaluation of RDFGraphGen. Finally, Section 6 concludes with summary and future work.

2 Related Work

Generating synthetic RDF data is not a new topic as it has been of interest to the research community for quite a long time. In this section, we provide an overview of existing generators of synthetic RDF datasets.

Tab2KG [11] is a method that is used for interpretation of tables with previously unseen data and automatically infers their semantics to transform them into semantic data graphs. The Tab2KG algorithm transforms tabular data into a semantic data graph by automatically inferring the domain ontology and mapping the table columns to the ontology classes and properties, before transforming the rows of the table into RDF triples. In short, the tabular data is provided, and the ontology is adjusted to fit this data. In contrast, RDFGraphGen uses explicit rules and constraints — it takes a description of a target data graph as a SHACL shapes graph and generates entities according to this description, using random or structured values for the objects in the RDF triples of the generated entities.

GAIA [14] is a generic RDF data generator that allows users to generate RDF triples by conforming to an ontology. It is OWL-based and generates RDF objects based on any correctly defined OWL ontology. The generator correctly generates a user-defined number of objects following the OWL ontology but offers no way to constrain the objects' values beyond datatype. RDFGraphGen uses a SHACL shapes graph as a description of the entities that should be generated, allowing the user to describe the object's values in great detail. RDFGraphGen also allows using properties from multiple ontologies in a generated entity since it does not generate data based on a specific ontology. The ontology is implicitly defined in SHACL shapes via the entity classes and the specified properties.

GRR [5] is a system for generating random RDF data using SPARQL-like syntax to describe the desired ontology. GRR can generate entities using the desired ontology and allows the user to provide input for the objects' values in the triples. However, GRR offers no method for constraining these values beyond providing them beforehead, making the process much more complicated for the user. RDFGraphGen in contrast uses the input SHACL shapes graph to constrain the values of the objects in the generated RDF triples.

PyGraft [13] is another tool designed for the creation of highly customized, domain-independent RDF datasets and RDF schemas. It takes input RDF schemas and uses a set of rules and constraints to generate synthetic RDF triples that are compliant with the input structure. It has the ability to simulate realistic data distributions, including edge and node properties, while ensuring that the generated data adheres to the input ontology's constraints. PyGraft is scalable, which makes it suitable for large datasets, and it is capable of handling various RDF vocabularies to produce diverse datasets that closely mimic real-world RDF data. RDFGraphGen, on the other hand, relies on the level of details and constraints contained in the input SHACL shapes graph to generate the desired synthetic RDF knowledge graph.

GenACT [15] is a data generator for temporal and evolving RDF graphs in the domain of social media activity during academic events — more specifically, academic conference tweets (ACT). This approach tries to address the gap of missing real-world knowledge graphs with specific capabilities that are necessary for the use cases in this domain. In contrast, RDFGraphGen is domain-independent and tries to solve the same gap from a more generalized perspective.

3 RDFGraphGen

In this section, we describe the design of RDFGraphGen, we explain how the graph structure and the literal values are generated and how to use the tool to generate RDF data from SHACL shapes.

RDFGraphGen has a straightforward workflow (Figure 1). Given a SHACL shapes graph as input, RDFGraphGen outputs RDF data. The size of the generated RDF graph is controlled by an input parameter: *scale-factor*. We also explain how the scale factor affects the number of entities and RDF triples.

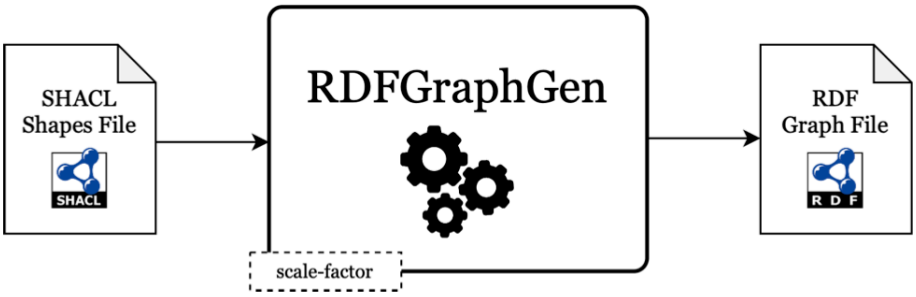

Figure 1: RDFGraphGen Workflow Diagram

3.1 Graph Generation Algorithm

The goal is to generate realistic and useful RDF data. For the input shapes graphs, RDFGraphGen generates a data graph that conforms to them. The aim is to create an algorithm that uses the structure of the input SHACL shapes and generates data following this structure. The goal is to use as many details from the shapes graph in order to populate the generated RDF data graph, even if the shapes are contradictory. Generally, the task of generating conforming data graphs is non-trivial[3].

3.1.1 Preliminaries on SHACL

To provide sufficient background, we start by introducing SHACL [2]. In general, the main purpose of SHACL is to define a set of constraints on nodes called *shapes*, which in turn are used to define constraints on an RDF graph. The set of shapes is referred to as a *shapes graph*[4] while the data that is checked is called the *data graph*. When the data graph satisfies all the shapes from the shapes graph, we say it *conforms*. A node in the data graph that must satisfy a shape is called the *focus node*. Focus nodes for a given shape are specified by *target declarations*.

Conceptually, there are two different types of shapes: *node shapes*, which describe constraints on the focus nodes themselves, and *property shapes* describing constraints on a set of values reachable from the focus node by a specified property. Structurally, a node shape $s = (n, C, T)$ consists of three components: (1) a shape name n, typically an IRI, (2) a conjunction of constraint components C, and (3) a (possibly empty) set of target declarations T. When there are target declarations, the shape can be considered a constraint on the data graph. Similarly, a property shape $p = (n, C, T, E)$ consists of all the components of a node shape, with an additional element: the *property path* E. For our purposes, the property path is always a predicate path, i.e., an IRI.

A *constraint component* describes the specific conditions that a focus node must satisfy. Node shapes can be considered as a conjunction of constraint components — all constraint components must be satisfied for the focus node. In particular, shape-based constraint components[5] allow for creating a network of shapes that can be used to inform the structure of the generated data graph.

3.1.2 Generating the Graph

The algorithm for generating RDF data is primarily based on the set of constraint components. The shape name and the target declarations are only used as additional information. We assume that the shapes graph is well-formed[6] and non-recursive[7], i.e., that it is syntactically correct SHACL and there are no self-referential shapes.

The algorithm is designed as follows. Given an input SHACL shapes graph S, and the number of desired entities m, RDFGraphGen generates an RDF graph G that contains m entities per explicitly defined node shape s in S. Specifically, we generate m entities for each node shape s, such that the triple $(s$, rdf:type, sh:NodeShape$)$ appears in the shapes graph S.

[3] Because SHACL corresponds to a logic [6], generating data corresponds to finding an model for a given SHACL theory. Even for perceived "simple" shapes, the task might be best solved by resorting to first-order logical reasoners.

[4] It is called a *graph* because SHACL syntax is defined in RDF itself.

[5] https://www.w3.org/TR/shacl/#core-components-shape

[6] https://www.w3.org/TR/shacl/#dfn-well-formed

[7] https://www.w3.org/TR/shacl/#shapes-recursion

Algorithm 1 GENERATEFROMNODESHAPE(S, s, e)

$G = \emptyset$
$s = (s_n, C, s_T)$
for each constraint component $c \in C$ **do**
  if c is a property shape component **then**
    Let p be the property shape associated with c.
    $G', E :=$ GENERATEFROMPROPSHAPE(S, p)
    Let r the predicate path associated with p.
    $G := G \cup G' \cup \{(e, r, e') \mid e' \in E\}$
  else if c is a node constraint component **then**
    Let s' be the shape from S associated with c.
    $G := G \cup$ GENERATEFROMNODESHAPE(S, s', e)
  else if c is a logical constraint component **then**
    $G := G \cup$ GENERATELOGICAL(S, c, e)
  end if
end for
return G

Algorithm 2 GENERATEFROMPROPSHAPE(S, p)

$G := \emptyset$
$p = (p_n, C, p_T, q)$
card := *determineCardinality*(C)
if q is a supported schema.org predicate **then**
  return G, *createSchemaValues*(q, card)
else if $\exists c \in C$ such that c is a datatype constraint component **then**
  Let datatype be the datatype associated with c.
  return G, *createLiteralValues*$(\mathsf{datatype}, \mathsf{card})$
end if
Let $E := \{e_1, \ldots, e_{\mathsf{card}}\}$ be a set of newly created IRIs.
if $\exists c \in C$ such that c is a property shape component **then**
  Let $C' \subseteq C$ be the set of property shape components from C.
  Let S' be the set of all property shapes associated with C'.
  for each $s' \in S'$ **do**
    Let q be the predicate path associated with s'.
    $G', E' :=$ GENERATEFROMPROPSHAPE(S, s')
    $G := G \cup G' \cup \bigcup_{e \in E} \{(e, q, e') \mid e' \in E'\}$
  end for
else if $\exists c \in C$ such that c is a node constraint component **then**
  Let $C' \subseteq C$ be the set of node constraint components from C.
  Let $S' = \{s'_1, \ldots, s'_k\}$ be the set of all k node shapes associated with C'.
  Partition E into mutually disjoint sets $E_1, \ldots, E_k$
  for each $s'_i \in S'$ **do**
    $G = G \cup \bigcup_{e \in E_i}$ GENERATEFROMNODESHAPE(S, s'_i, e)
  end for
else if $\exists c \in C$ such that c is a logical constraint component **then**
  Let $C \subseteq C'$ be the set of logical constraint components from C.
  $G := \bigcup_{c' \in C'} \bigcup_{e \in E}$ GENERATELOGICAL(S, c', e)
end if
return G, E

We focus our explanation on the generation of a single entity e for a node shape s. Recall that a shape contains a set of constraint components C. The procedure GENERATEFROMNODESHAPE(S, s, e) is described in Algorithm 1. It uses the shapes graph S, a node shape s, and an IRI e to generate triples describing entity e using s, possibly referring to other shapes from S, and outputs a synthetic RDF graph G. It looks at every supported constraint component and generates data accordingly. For node constraint components (`sh:node`), it simply generates some data for every component, for entity e. Similarly, for the supported logical constraint components `sh:and`, `sh:or`, and `sh:xone`, we select an approprate subset of shapes for which data is generated w.r.t. e. (see Algorithm 3).

Algorithm 3 GENERATELOGICAL(S, c, e)

$S' := \emptyset$
if c is an 'and' constraint component **then**
  S' is the set of shapes from S associated with c.
else if c is a 'or' constraint component **then**
  S' is a random, non-empty subset of the shapes in S associated with c.
else if c is a 'xone' constraint component **then**
  S' is a set containing exactly one random shape from S associated with c.
end if
return $\bigcup_{s' \in S'}$ GENERATEFROMNODESHAPE(S, s', e)

Example 1 *Consider the shape* `<MyShape>`*:*

```
<myShapeA> a sh:NodeShape ;
  sh:node <myShapeB>, <myShapeC> ;
  sh:or ( <myShapeD> <myShapeE> ) .
```

where we assume `<myShapeB>, ...,<myShapeE>` *are defined. The generator creates the IRI e and calls* GENERATEFROMNODESHAPE(S, s, e) *for* $s =$ `<myShapeA>` *(initially),* $s =$ `<myShapeB>` *and* $s =$ `<myShapeC>` *(all node constraint components), and also, randomly, for* $s =$ `<myShapeD>` *or* $s =$ `<myShapeE>` *or both (logical constraints).*

Finally, and most importantly, data is generated for every associated property shape (`sh:property`) p, handled by the procedure GENERATEFROMPROPSHAPE(S, p) described in Algorithm 2, which returns a set of new RDF terms E, and (a possibly empty) RDF graph G' representing triples that are recursively generated for entities from E. This is the most crucial procedure, as the property shapes indicate the structure of the generated RDF graph by describing the immediate neighborhood.

Example 2 *Continuing from Example 1, we can add the following additions to the shape definition:*

```
<myShapeA> sh:property <myPropShape> .
<myPropShape> sh:path <pred> ; sh:minCount 1 .
```

This means that the generator calls GENERATEFROMPROPSHAPE(S, p)*, where* $p =$ (`<myPropShape>`, {`sh:minCount 1`}, $\emptyset$, `<pred>`)

The procedure works as follows. First, we determine how many values are generated for the property shape (indicated by *determineCardinality*(C)) based on the keywords `sh:minCount` and `sh:maxCount`. Essentially, a random integer is selected between the minimum and maximum cardinality. If there is an inconsistency (i.e., the minimal cardinality is greater than the maximal), the minimal indicated cardinality is followed.

Next, we distinguish between generating literal values, or new entities. If the predicate path is a schema.org predicate, we generate an appropriate value with the helper function *createSchemaValues*$(q,$ `card`$)$. Essentially, `card` indicates the number of values to be generated appropriate to q. Similarly, if the values are intended to be of a certain datatype (by default `xsd:string` values), these are handled accordingly by the helper function *createLiteralValues*(`datatype`, `card`). We elaborate on both helper functions in Section 3.2.

Finally, if we expect the values for the property to be other entities, e.g., referring to other shapes with `sh:node`, `sh:property`, or a logical constraint component, then the generation procedure is called recursively in accordance with the indicated structure of the shape.

Example 3 *Recall* `<myPropShape>` *from Example 2. The generator recognizes that at least one value must be generated, so it will choose* `card` ≥ 1*, for example* `card` $= 2$*. Next, because* `<pred>` *is not a specially supported predicate path and because there is no datatype explicitly defined, it will generate two random strings* $\texttt{str}_1$ *and* $\texttt{str}_2$*, and adds the triples (e,* `<pred>`*,* $\texttt{str}_1$*) and (e,* `<pred>`*,* $\texttt{str}_2$*) to the output graph.*

3.1.3 Size of the Generated RDF Graph

RDFGraphGen uses a scale factor to determine the size of the generated RDF dataset. The value of the parameter corresponds to the number of entities that will be generated from all top-level node shapes in the input SHACL graph.

By *top-level* node shapes we mean node shapes that are either stand-alone and not related to other node shapes from the SHACL graph, or that only appear as subjects in the predicates pointing to other node shapes from the SHACL graph. The other node shapes, which solely appear as objects in predicates from other node shapes, are generated in numbers depending on the cardinality of the same predicate, as described in Algorithm 2.

Example 4 *Consider the shape* `<myShapeF>`*:*

```
<myShapeF> a sh:NodeShape ;
    sh:property [ sh:path <predicate> ;
        sh:node <myShapeG> ;
        sh:minCount 2 ;
        sh:maxCount 4 ] .
```

This means that `<myShapeF>` *is a top-level shape, connected via* `<predicate>` *to the non top-level shape* `<myShapeG>`*.*

If the scale-factor is set to 100, the generator will generate 100 entities based on `<myShapeF>`*, and between 200–400 entities based on* `<myShapeG>` *(i.e., 2 to 4 per entity based on* `<myShapeF>`*), resulting in a total of 300–500 entities in the generated RDF graph.*

3.2 Generating Literal Values

To generate *useful* RDF data, we generate meaningful literal values whenever possible. Recall that Algorithm 2 explicitly refers to schema.org for the creation of values, based on the predicate path of the property shape. The focus on schema.org is not a fundamental limitation: our implementation can easily be extended to include other vocabularies in the future.

If we need to generate a value that is not related to schema.org, then the values are are generated based on the specified datatype (`sh:datatype`), and if none is given, we default to `xsd:string`.

3.2.1 Generating Schema.org Literals

When generating a new literal, the first step is to try and infer what would be its logical value, given the predicate and the RDF type of the entity as defined in the input SHACL shape. For example, if a triple is being generated for an entity of type `schema:Person` from the widely used schema.org vocabulary [12], where the predicate is `schema:firstName`, it is natural that the generated value of the object should be a human first name. In this case, the generator uses a set of first names to pick one randomly. Even further, if the `schema:Person` entity has a `schema:gender` value of either male or female, the generator will specifically use the list of male or female first names, accordingly, to pick one at random.

The generator contains sets of collected values for the most common properties from schema.org. These sets are used by RDFGraphGen to randomly select a value for such properties, when they appear in the SHACL shapes. More specifically, RDFGraphGen has collections of male and female names and surnames, job titles, addresses and street names, book titles, book genres, TV series titles, movie titles, movie genres, etc., that have been collected from online resources. This feature of RDFGraphGen can easily be extended in the future to include many more domains.

Additionally, the generator contains several rules about constructing values for other well-known schema.org predicates, e.g., a full name or an email address. In scenarios where the input SHACL shape uses `schema:name`, the value is constructed by concatenating a random (gendered) first name and a last name. When the predicate `schema:email` is used without a specified pattern, RDFGraphGen generates an email address in the form `firstname-lastname@gmail.com`.

If the predicate is unknown, it can still provide useful information to determine a suitable value. For example, if the name of the predicate is `birthDate`, from a random or unknown vocabulary, the term `date` suggests that the object should be a date. This will be caught by RDFGraphGen and a date value will be generated. Similar behavior is implemented for other predicates from unknown vocabularies, such as `telephone`, `phone`, `email`, etc.

3.2.2 Generating Generic Values

When the value cannot be picked from a pre-defined set of specific values, we generate it randomly. This random generation step uses all of the constraints components defined in the input SHACL shape, e.g., datatype, minimum length, maximum length, pattern, minimum and maximum value, etc.

This is what makes the RDFGraphGen generator general-purpose and domain-independent: it can generate synthetic RDF triples and graphs from any domain (life sciences, social networking, entertainment, linguistics, geography, etc.), regardless of whether it is explicitly familiar with it or not.

3.3 Usage

To make the RDFGraphGen generator more easily available for the community, we packaged it as a Python library and published it online on PyPi[2]. RDFGraphGen is available via the command: `pip install rdf-graph-gen`, which downloads and installs it locally on the user's machine. Afterwards, RDFGraphGen can be used via the command-line, using the command:

```
rdfgen input-shapes.ttl output-graph.ttl scale-factor
```

The command `rdfgen` takes three arguments as input:

- the file containing the input SHACL shapes graph, e.g., `input-shapes.ttl`,
- the file that the generated synthetic RDF graph is to be written to, e.g., `output-graph.ttl`, and
- the scale factor that determines the size of the generated RDF graph (`scale-factor`).

We optimized RDFGraphGen for generating large graphs — data is generated in batches that are serialized in the output file when the batch size is reached, in order to continuously free up main memory. We also implemented concurrency to parallelize and speed up data generation for large graphs.

The full RDFGraphGen code, along with the CSV files containing pre-defined sets of values, examples of SHACL shape files and generated graph examples, are publicly available on GitHub[1]. Using a CI/CD pipeline based on GitHub actions, we release a new version of the RDFGraphGen generator on PyPi when there are changes in the GitHub repository.

4 Example Use Case

To showcase how RDFGraphGen works for specific SHACL shapes, this section presents an example of using the `schema:Person` class from schema.org. Other examples, from various domains using different ontologies are available on the project's GitHub page[1].

In this example, the input SHACL shapes graph (Listing 1) describes entities that are of type `schema:Person`. From the definition, each such entity is constrained to have a given name and a last name, or a full name; exactly one date of birth, which must be earlier then the potential death date; a gender with one of the available values; an email with a value constrained by a pattern; another using the `schema:email` property; a telephone number; a job title; an address, which has a specified complex type (it is an object, not a literal). Additionally, the address objects have a street address that is a string value; a postal code, which can be a string or an integer with values between 10,100–10,999.

Listing 1. Input SHACL Shapes Graph

```
ex:PersonShape a sh:NodeShape ;
    sh:targetClass schema:Person ;
    sh:xone ( [ sh:property [
                    sh:path schema:givenName ;
                    sh:datatype xsd:string ;
                    sh:minCount 1 ;
                ] ;
                sh:property [
                    sh:path schema:familyName ;
                    sh:datatype xsd:string ;
                    sh:minCount 1
                ]]
              [ sh:path schema:name ;
                sh:datatype xsd:string  ;
                sh:minCount 1 ] );
    sh:property [
        sh:path schema:birthDate ;
```

```
        sh:lessThan schema:deathDate ;
        sh:minCount 1 ;
        sh:maxCount 1 ] ;
    sh:property [
        sh:path schema:gender ;
        sh:in ( "female" "male" ) ;
        sh:maxCount 1 ] ;
    sh:property [
        sh:path ex:workMail ;
        sh:pattern "^[a-z0-9]+\\.[a-z0-9]+@work\\.[a-z]{2,3}$"
        ] ;
    sh:property [
        sh:path schema:email ;
        sh:maxCount 1 ] ;
    sh:property [ sh:path schema:telephone ] ;
    sh:property [ sh:path schema:jobTitle ] ;
    sh:property [
        sh:path schema:address ;
        sh:node ex:AddressShape ;
        sh:minCount 1 ] .

ex:AddressShape a sh:NodeShape ;
    sh:class schema:PostalAddress ;
    sh:property [
        sh:path schema:streetAddress ;
        sh:datatype xsd:string ; ] ;
    sh:property [
        sh:path schema:postalCode ;
        sh:or ( [ sh:datatype xsd:integer ;
                  sh:minInclusive 10100 ;
                  sh:maxInclusive 10999  ]
                [ sh:datatype xsd:string ] ) ] .
```

After running the RDFGraphGen generator on the input SHACL shapes file and setting the parameter `scale-factor` to 2, we get the generated synthetic RDF graph (Listing 2).

Listing 2. Output RDF Graph

```
<http://example.org/ns#Node100> a schema:Person ;
    ex:workMail "18b.dqjz8szqwne7nm@work.ekx" ;
    schema:address <http://example.org/ns#Node101> ;
    schema:birthDate "1965-07-07"^^xsd:date ;
    schema:deathDate "1989-07-07"^^xsd:date ;
    schema:email "barton_aldridge@gmail.com" ;
    schema:familyName "Aldridge" ;
    schema:gender "male" ;
    schema:jobTitle "bartender" ;
    schema:givenName "Barton" ;
    schema:telephone "647-466-552849" ;
    sh:description ex:PersonShape .

<http://example.org/ns#Node102> a schema:Person ;
    ex:workMail "sqq2.s7ojq@work.vab" ;
    schema:address <http://example.org/ns#Node103> ;
    schema:birthDate "1986-07-07"^^xsd:date ;
    schema:email "sarajanebenjamin@gmail.com" ;
    schema:gender "female" ;
    schema:jobTitle "psychologist" ;
    schema:name "Sarajane Benjamin" ;
    schema:telephone "722-279-0247032" ;
    sh:description ex:PersonShape .

<http://example.org/ns#Node101> a schema:PostalAddress ;
    schema:postalCode 10481 ;
```

```
27      schema:streetAddress "no. 3 Lily st" ;
28      sh:description ex:AddressShape .
29
30  <http://example.org/ns#Node103> a schema:PostalAddress ;
31      schema:postalCode "6plO65qAPywG" ;
32      schema:streetAddress "no. 1 Gillette ave" ;
33      sh:description ex:AddressShape .
```

When generating the synthetic RDF graph, given this input, RDFGraphGen produces an output such as the one presented in Listing 2. When it generates the name of each person entity, only one option from the `sh:xone` constraint can be selected (Line 3, Listing 1). For the first person (Line 1, Listing 2), the generator has created separate predicates for the given name (Line 10, Listing 2) and the family name (Line 7, Listing 2). For the second person (Line 14, Listing 2), the generator has generated a single predicate for the full name (Line 21, Listing 2), and the full name includes both the given name and the family name as a single value.

Furthermore, predicates related to dates have a date value for the object despite not explicitly containing a `sh:datatype` constraint in the description. The generator extracts information from the predicate names of `schema:birthDate` and `schema:deathDate` (Lines 17-18, Listing 1), determines that the objects' values in the generated RDF triples should be dates, and then generates them based on the constraints specified in the input SHACL shapes graph.

The email address values (`schema:email`) consist of the given name and the family name of the person (Lines 6 and 18, Listing 2). This has been specifically predefined in the generator when working with `schema:Person` entities and their email addresses. The other email address value (`ex:workMail`) has a definition that is not part of schema.org and therefore its value is constrained by the pattern in the SHACL graph (Line 27, Listing 1). The generator produces values that conform to the defined pattern (Lines 2 and 15, Listing 2).

The generated phone numbers follow a specific pattern. This is again based on the predicate name, `schema:telephone` (Line 32, Listing 1) and therefore the generator applies the predefined pattern constraint (Lines 11 and 22, Listing 2).

The address for each person is generated as a separate RDF entity (Lines 25 and 30, Listing 2) based on the SHACL shape in the input file, which constrains the address. Given that there are no explicit cardinality restrictions on the relation between people and their addresses, each person has exactly one address. So, given that `scale-factor` was set to 2 when starting the generator, the output contains a total of 4 entities: 2 person entities (as top-level shapes) and 2 address entities (as shapes directly connected to a top-level shape).

	Single Simple Shape		Single Complex Shape		Three Simple Shapes		Three Complex Shapes	
Scale Factor	RDF Triples	Time (s)	RDF Triples	Time (s)	RDF Triples	Time (s)	RDF Triples	Time (s)
1	2	4	9	4	8	4	13	4
10	20	4	89	4	80	4	135	4
100	200	4	862	4	800	5	1,401	5
1,000	2,000	5	8,477	5	8,000	5	13,939	5
10,000	20,000	8	84,975	8	80,000	8	139,969	9
100,000	200,000	10	850,313	24	800,000	16	1,400,109	30
1,000,000	2,000,000	35	8,504,410	164	8,000,000	102	14,004,068	244
10,000,000	20,000,000	287	85,042,729	1,608	80,000,000	1,352	140,044,876	2,335
100,000,000	200,000,000	2,871	850,497,903	17,312	800,000,000	9,888	1,371,272,422	22,734

Table 1: Performance Evaluation of RDFGraphGen

5 Performance Evaluation

6 Conclusion and Future Work

In this paper, we introduced RDFGraphGen — a general-purpose, domain-agnostic synthetic RDF graph generator based on SHACL shapes. To the best of our knowledge, this is the first RDF generator based on SHACL constraints. The synthetic RDF knowledge graphs generated by RDFGraphGen can be used in many different scenarios in the software development cycle: application testing, algorithm testing, application benchmarking, software quality control, training of machine learning models, etc.

RDFGraphGen is domain-independent: the generated RDF graph contains data from the domain that is described in the source SHACL shapes file. Additionally, RDFGraphGen has the ability to use pre-collected values for schema.org classes and predicates and can be extend to include values for any ontology. This allows for the generator to be fine-tuned and adapted to the needs of specific use cases and domains.

In the future, we plan to extend the support for human-friendly literal values for an increased number of ontologies, classes, and properties in a manner similar to what we did with a subset of schema.org. To make our finding reproducible, and enable others to easily use RDFGraphGen, we have published it as open-source, under the MIT license.

References

[1] RDF 1.1 Concepts and Abstract Syntax. https://www.w3.org/TR/rdf11-concepts/.

[2] Shapes Constraint Language (SHACL). https://www.w3.org/TR/shacl/.

[3] Web Data Commons - RDFa, Microdata, Embedded JSON-LD, and Microformats Data Sets - October 2024. http://webdatacommons.org/structureddata/2024-12/stats/stats.html.

[4] Renzo Angles, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. The Linked Data Benchmark Council: A Graph and RDF Industry Benchmarking Effort. *ACM SIGMOD Record*, 43(1):27–31, 2014.

[5] Daniel Blum and Sara Cohen. Generating RDF for Application Testing. In *9th International Semantic Web Conference ISWC 2010*, page 105, 2010.

[6] Bart Bogaerts, Maxime Jakubowski, and Jan Van den Bussche. SHACL: A Description Logic in Disguise. In *Logic Programming and Nonmonotonic Reasoning*, pages 75–88, 2022.

[7] Georg Buchgeher, David Gabauer, Jorge Martinez-Gil, and Lisa Ehrlinger. Knowledge Graphs in Manufacturing and Production: A Systematic Literature Review. *IEEE Access*, 9:55537–55554, 2021.

[8] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets. In *SIGMOD*, pages 145–156, 2011.

[9] Kleanthi Georgala, Mirko Spasić, Milos Jovanovik, Henning Petzka, Michael Röder, and Axel-Cyrille Ngonga Ngomo. MOCHA2017: The Mighty Storage Challenge at ESWC 2017. In *Semantic Web Challenges*, pages 3–15, 2017.

[10] Kleanthi Georgala, Mirko Spasić, Milos Jovanovik, Vassilis Papakonstantinou, Claus Stadler, Michael Röder, and Axel-Cyrille Ngonga Ngomo. MOCHA2018: The Mighty Storage Challenge at ESWC 2018. In *Semantic Web Challenges*, pages 3–16, 2018.

[11] Simon Gottschalk and Elena Demidova. Tab2KG: Semantic Table Interpretation With Lightweight Semantic Profiles. *Semantic Web*, 13(3):571–597, 2022.

[12] Ramanathan V Guha, Dan Brickley, and Steve Macbeth. Schema.org: Evolution of Structured Data on the Web. *Communications of the ACM*, 59(2):44–51, 2016.

[13] Nicolas Hubert, Pierre Monnin, Mathieu d'Aquin, Davy Monticolo, and Armelle Brun. PyGraft: Configurable Generation of Synthetic Schemas and Knowledge Graphs at Your Fingertips. *arXiv*, 2024. URL https://arxiv.org/abs/2309.03685.

[14] Tanguy Raynaud, Samir Amir, and Rafiqul Haque. A Generic and High-Performance RDF Instance Generator. *International Journal of Web Engineering and Technology*, 11(2):133–152, 2016.

[15] Gunjan Singh, Udit Arora, Shashikant Kumar, Riccardo Tommasini, Pieter Bonte, Sumit Bhatia, and Raghava Mutharaju. GenACT: An Ontology-Based Temporal Web Data Generator. In *International Conference on Conceptual Modeling*, pages 317–336. Springer, 2024.