

---

# RDFGRAPHGEN: A SYNTHETIC RDF GRAPH GENERATOR BASED ON SHACL CONSTRAINTS

---

A PREPRINT

**Marija Vecovska, Milos Jovanovik**

Faculty of Computer Science and Engineering  
Ss. Cyril and Methodius University in Skopje, N. Macedonia  
marija.vecovska@students.finki.ukim.mk  
milos.jovanovik@finki.ukim.mk

July 26, 2024

## ABSTRACT

This paper introduces RDFGraphGen, a general-purpose, domain-independent generator of synthetic RDF graphs based on SHACL constraints. The Shapes Constraint Language (SHACL) is a W3C standard which specifies ways to validate data in RDF graphs, by defining constraining shapes. However, even though the main purpose of SHACL is validation of existing RDF data, in order to solve the problem with the lack of available RDF datasets in multiple RDF-based application development processes, we envisioned and implemented a reverse role for SHACL: we use SHACL shape definitions as a starting point to generate synthetic data for an RDF graph. The generation process involves extracting the constraints from the SHACL shapes, converting the specified constraints into rules, and then generating artificial data for a predefined number of RDF entities, based on these rules. The purpose of RDFGraphGen is the generation of small, medium or large RDF knowledge graphs for the purpose of benchmarking, testing, quality control, training and other similar purposes for applications from the RDF, Linked Data and Semantic Web domain. RDFGraphGen is open-source and is available as a ready-to-use Python package.

## 1 Introduction

The Semantic Web has been a topic of interest for academia and the industry for over two decades now [5]. Over this period, we have witnessed the acceptance of RDF [1] as a standard for representing and publishing data on the Web and internally in organizations, along with a plethora of other W3C standards, such as RDFS, OWL, SPARQL, etc. [12]. These standards, most notably RDF, have been widely accepted on the Web, especially after the introduction of Schema.org as a lightweight ontology / vocabulary [11]. Currently, around 50% of the webpages on the Web contain RDF-based metadata [3].

This has also led to the development of many tools, libraries, applications, systems and databases which work with or use RDF as a model for the data. During the development lifecycle of each of them, 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 sometimes require application-specific or domain-specific RDF datasets, which are not always readily available. This is where synthetic RDF datasets come in. They are datasets which contain entities and data which are artificial, but follow the structure and vocabulary or ontology expected from the application or the system. They fulfill the need for data in a specific format, from a specific domain, for an application or a system to be tested or benchmarked [8][9]. Depending on the specific needs, these synthetic RDF datasets can contain from a few, up to millions of different entities and RDF triples [7].

Synthetic RDF data can be created using data generators. These are usually task-specific, i.e. are created for the specific purpose at hand, when benchmarking or testing a system or an application. What was currently missing, was a generic synthetic RDF data generator, which would be able to generate artificial RDF data from any domain.

In order to fill this missing piece in the puzzle of the RDF world, we designed and implemented a general-purpose, domain-independent synthetic RDF data generator, based on SHACL constraints. It reverses the role of the Shapes Constraint Language (SHACL), which was designed for validation of existing RDF data; we now use SHACL shapes as a starting point to generate new, artificial RDF data and construct an RDF knowledge graph of the desired size.

SHACL is a W3C standard for validating RDF graphs against a set of conditions [2]. These conditions are provided as shapes defined in RDF (shape graph), which are used to validate that a given RDF graph (data graph) satisfies a set of conditions. However, these shapes can also be viewed as a description of the RDF graphs that do satisfy their conditions. We use this characteristic of SHACL shapes to generate synthetic RDF graphs which satisfy the constraints defined in the shapes.

This paper sets to describe the design and implementation of RDFGraphGen - a domain-independent generator of synthetic RDF graphs based on SHACL constraints. The SHACL constraints are used as a description of the structure and data composing the RDF graph, and in accordance with these rules, the corresponding RDF triples are generated. RDFGraphGen is domain-agnostic, meaning that the source SHACL shapes can be from any domain. The generator can generate an RDF graph with a specified number of entities, providing flexibility for the end-users. These small, medium or large RDF datasets (knowledge graphs), generated by RDFGraphGen, can then be used in benchmarking, testing, quality control, training and other similar tasks in the application lifecycle in the domains of RDF, Linked Data and the Semantic Web. Additionally, RDFGraphGen is open-source [20] and is available as a ready-to-use Python package [21].

## 2 Related Work

Generating synthetic RDF data is not a new topic. It has been of interest to the research community for quite a long time, and here we present a discussion on the existing solutions and how our approach relates to them. After that, we present a short overview of SHACL shapes.

### 2.1 Synthetic RDF Data Generation

Tab2KG 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 [10]. The Tab2KG algorithm transform 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. This method, despite generating large volumes of RDF data, does not support generating data graphs from a particular user-defined ontology, nor does it allow defining rules about the generated data. In other words, the tabular data is provided, and the ontology is adjusted to fit this data. In contrast, RDFGraphGen uses a reversed approach to generate data, i.e. it takes a description of a target data graph as a SHACL shape, and generates entities according to this description, using random or structured values for the objects in the RDF triples of the generated entities.

GAIA is a generic RDF data generator that allows users to generate RDF triples by conforming to any ontology [17]. It is OWL-based, and generates RDF objects based on any properly defined OWL ontology. After testing it, it is apparent that the generator correctly generates a user defined number of objects following the OWL ontology. However, GAIA offers no way to constrain the objects' values beyond datatype. RDFGraphGen uses a properly defined SHACL shape as a description of the entities that should be generated, allowing the user to describe the object's values in great detail, and the generator uses this details to generate more logically correct entities. RDFGraphGen also allows using properties from multiple ontologies in a generated entity, since it doesn't generate data based on a specific ontology. The ontology is implicitly defined in the SHACL shape via the properties. However, RDFGraphGen is SHACL-based, leading to worse interconnectivity of the generated entities and worse control over the number of generated entities for each class.

GRR is a system for generating random RDF data, using SPARQL-like syntax to describe the desired ontology [6]. GRR can generate entities using the desired ontology, and it allows the user to provide input for the object's values. However, GRR offers no method for constraining the objects' values beyond providing them beforehand, making the process much more complicated for the user. GRR is much more suitable for generating networks of interconnected entities, according to the user's definition, which is beyond the capabilities of RDFGraphGen at this time.

Our research team has some experience in designing and using RDF data generators in several domains, as well. For instance, in the domain of social network data, for the purpose of benchmarking RDF storage solutions, we have developed a domain-specific RDF dataset generator [19]. It is written in the Java programming language, and builds on a previous generator, in order to improve some of the metrics in the resulting graph and make its features closer to a real-world RDF dataset. Aside from this, our team has also worked with other RDF graph generators, for instance in

the field of geo-spatial data [16][15][14] and in benchmarking RDF storage solution [13][18]. All of these examples include purpose-built RDF data generators, which serve a specific need. In contrast, our approach with RDFGraphGen is to provide a general-purpose, domain-independent generator which can work for any scenario.

## 2.2 A Brief Overview of SHACL Shapes

SHACL (Shapes Constraint Language) is a W3C standard used to validate RDF data against a set of conditions, known as shapes, ensuring the data conforms to specific requirements [2]. It enables the definition of constraints on RDF graphs, enabling validation and verification of data in a structured and standardized way.

A SHACL shape determines how to validate a focus node based on the values of properties and other characteristics of the focus node. For example, shapes can declare a condition that a focus node has a particular value for a given property, along with a minimum number of values for the property. The shapes are written in RDF, as well, forming SHACL graphs.

SHACL defines two types of shapes:

- **Node shapes:** shapes about the focus node itself,
- **Property shapes:** shapes about the values of a particular property or path for the focus node.

A single SHACL graph can contain multiple node shapes. Each node shape usually contains multiple property shapes.

Below we show an example of a SHACL shape with a focus node that represents a person.

---

Example 1: A SHACL Shape for Person Entities

---

```
@prefix sh: <http://www.w3.org/ns/SHACL#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.com/ns#> .

ex:PersonShape
  a sh:NodeShape ;
  sh:targetClass ex:Person ;
  sh:property [
    sh:path ex:name ;
    sh:datatype xsd:string ;
    sh:maxCount 1;
    sh:name "Person's name" ;
  ] ;
  sh:property [
    sh:path ex:birthDate ;
    sh:lessThan ex:deathDate ;
    sh:maxCount 1 ;
  ] ;
  sh:property [
    sh:path ex:gender ;
    sh:in ( "female" "male" ) ;
  ] .
```

---

The SHACL shapes graph in this example contains one nodes shape, `ex:PersonShape`. This shape targets any `ex:Person` entity in the RDF data graph. The node shape contains three property shapes which describe the properties of the target node. The first property shape constraints the person's name, the second constraints its birth date and the third its gender. A property shape describes a property in an RDF data graph. The property is specified as the object in the RDF triple where the property shape is the subject (usually a blank node), and `sh:path` is the predicate, i.e. `ex:name`, `ex:birthDate` and `ex:gender` in this case.

## 3 RDFGraphGen

RDFGraphGen is a generic RDF graph generator that generates RDF entities according to a provided SHACL shapes graph. RDFGraphGen is domain-agnostic, and can be used in multiple scenarios where non-specific RDF data is needed, such as application or algorithm testing, system and tool benchmarking, software quality control, training of

machine learning models, etc. In this section we will go into the details of how it was designed and implemented, and we will present several examples which showcase how it works in practice.

### 3.1 Design

SHACL shape graphs were introduced as a standardized way to validate that a given dataset of RDF data satisfies a given set of conditions. However, the same SHACL shape graphs can also be viewed as a description of the RDF data graphs that do satisfy these conditions [2]. RDFGraphGen uses these conditions as rules to generate synthetic data into RDF graphs, that conform to the description provided in the SHACL shape graphs.

First, let us look into the overall structure of the generator. The generator consists of an extracting component and a generating component. A SHACL shape graphs can describe one or multiple different data graphs. Each data graph is separately described by a SHACL node shape. These SHACL node shapes can contain multiple SHACL property shapes which describe the triples in the RDF data graphs.

First, given a SHACL shapes graph, the extracting component analyzes it, locates all of the SHACL node shapes described in it, and creates a shape map for each of them. A shape map contains the description of a specific SHACL node shape. Then, the generating component generates a specified number of RDF entities into an RDF data graph, using the shape maps as templates.

**The Generation Steps.** The RDFGraphGen is intended to be used in the following way: the user provides a Turtle (.ttl) file which contains a SHACL shapes graph (e.g. `input-shapes.ttl`), an empty Turtle file in which the generated RDF will be written into (e.g. `output-graph.ttl`), and the number of entities which should be generated (`entityCount`). After the generation process has finished, the generated RDF data graph can be accessed via the output file (e.g. `output-graph.ttl`).

The generation process consists of the following steps:

- Step 1: The generator gets, reads and parses the input file which contains the SHACL shapes graph.
- Step 2: Next, the generator identifies the SHACL node shapes.
- Step 3: The generator creates a shape map for each separate SHACL node shape. It stores all of these separate shape maps in a single root shape map.
- Step 4: The generator generates RDF entities with their corresponding properties and values as separate smaller RDF data graphs, based on the shape maps.
- Step 5: The generator writes out the generated RDF data graphs into a single RDF data graph, into the output file.

In the next section, we will present a more in-depth explanation of how the RDFGraphGen generator achieves this.

### 3.2 Implementation

In this section we will describe the implementation of the RDFGraphGen generator. RDFGraphGen was coded using the Python programming language, and it uses a few Python libraries: `rdflib` for RDF graph functionality and `exrex` for generating strings using regular expressions.

#### 3.2.1 Step 1 & 2: Finding the SHACL Node Shapes

Reading and parsing the input Turtle file in Python is trivial. The SHACL node shapes are usually the subject in an RDF triple where the predicate is the `rdf:type` predicate and the object is `sh:NodeShape`. RDFGraphGen selects there triples and extracts the node shapes and their definitions.

#### 3.2.2 Step 3: Creating a Shape Map

The next step is to extract the constraints for each node shape from the SHACL graph, to create a sub-dictionary for each separate shape, and to provide a simple way for accessing the options from the SHACL lists - used for logical constraint components and `sh:in`.

RDFGraphGen extracts the description of the target data graph from the corresponding SHACL node shape and its nested property shapes in a structure called a *shape map*. A shape map is a dictionary-like structure representation of a SHACL shape. The property shapes which are nested in a node shape, are also represented by shape maps and stored as properties in the shape map of their node shape.

To explain the procedure more precisely, we provide a list of definitions. The definitions relating to SHACL shapes are reiterated from the SHACL standard [2] in our context, while the others, relating to shape maps, are our own.

*Definition 1:* A SHACL node shape  $NS$  is a subject of an RDF triple where the predicate is `rdf:type` and the object is `sh:NodeShape`.

*Definition 2:* A SHACL property shape  $PS$  is the object of an RDF triple where the subject is an  $NS$  and the predicate is `sh:property`.

*Definition 3:* A SHACL shape  $S$  is either a SHACL node shape ( $NS$ ) or a property shape ( $PS$ ).

*Definition 4:* A SHACL shapes graph  $ShapesG$  is an RDF graph that contains SHACL shapes ( $S$ ). Each shapes graph contains  $n$  shapes:  $S_1, \dots, S_n$ .

*Definition 5:* A property  $P_i$  is the object of an RDF triple where the subject is a SHACL property shape  $PS_i$  and the predicate is `sh:path`.

*Definition 6:* We define  $SM_i$  as the shape map of the shape  $S_i$ .  $SM_i$  contains the constraints defined for  $S_i$  in the source SHACL shapes graph ( $ShapesG$ ).

*Definition 7:* We define  $SM\_properties$  as the structure in a shape map ( $SM$ ) that contains other shape maps ( $SM$ ).

*Definition 8:* A shape  $S_i$  contains a property shape  $PS_i$  if  $S_i$  is the subject of an RDF triple where the predicate is `sh:property` and the object is  $PS_i$ .

*Definition 9:* A shape  $S_i$  points to a shape  $S_j$  if there exists a shape  $S_k$ , such that  $S_i$  contains  $S_k$  directly or indirectly, and there exists an RDF triple where  $S_k$  is the subject, `sh:node` is the predicate and  $S_j$  is the object.

Algorithm 1 describes how a shape  $S$  is transformed into a shape map  $SM$ , while handling all of its nested shapes. This algorithm is recursive and is used for mapping all of the SHACL shapes from the source SHACL document, into their corresponding shape maps.

---

**Algorithm 1** Transform Shape into a Shape Map

---

**Input** A SHACL Shape ( $S$ ), and the SHACL shapes graph ( $ShapesG$ ) that contains it

**MapShape**( $S$ ,  $ShapesG$ )

**Create a**  $SM$

**For Each triple**  $S$  - predicate - object in  $ShapesG$

        // means that “object” is a Property Shape (PS) contained inside the Shape ( $S$ )

**If** predicate == `sh:property`

            Add **MapShape**(object,  $ShapesG$ ) to  $SM\_properties$ .

**Else**

            Add predicate : object pair to  $SM$

---

In Algorithm 1,  $ShapesG$  is a shapes graph which contains one or more shapes. Each node shape  $NS_1, \dots, NS_n$  in the  $ShapesG$  graph is transformed into a shape map  $SM_1, \dots, SM_n$  accordingly, and added to  $SM_0$ , as  $NS_i : SM_i$  key - value pairs. Here,  $SM_0$  represents the root shape map that represents the shapes graph.

Each node shape  $NS_i$  can contain zero or more property shapes  $PS_1, \dots, PS_n$ .

If a  $NS_i$  contains at least one  $PS$ , a structure  $SM\_properties$  is added to  $SM_i$ , to store the shape maps for each  $PS$ .

For each  $PS_j$  contained in a  $NS_i$ , a  $SM_j$  is added to  $SM\_properties$ , as a  $P_j : SM_j$  key - value pair, where  $P_j$  is the property described by  $PS_j$ .

If  $PS_j$  contains another property shape  $PS_k$ , the same procedure is repeated. If not, the SHACL constraints from  $P_j$  are added to  $SM_j$ .

The last 2 steps are repeated until there are no more nested property shapes.

To illustrate the procedure, let's use the SHACL shape from Example 1. Below is the root shape map for it, presented as Example 2.

---

Example 2: A Shape Map for the SHACL Shape from Example 1

---

```
{ 'ex:PersonShape':
  { 'rdf:type': 'sh:NodeShape',
    'sh:targetClass': 'ex:Person',
    'properties':
```

```

    { 'ex:birthDate':
      { 'sh:lessThan': 'ex:deathDate',
        'sh:maxCount': '1',
        'sh:path': 'ex:birthDate' },
      'ex:deathDate':
      { 'sh:path': 'ex:deathDate' },
      'ex:gender':
      { 'sh:in': [ 'female', 'male' ],
        'sh:path': 'ex:gender' },
      'ex:name':
      { 'sh:datatype': 'xsd:string',
        'sh:maxCount': '1',
        'sh:name': 'person's name',
        'sh:path': 'ex:name' }
    }
  }
}

```

### 3.2.3 Step 4 & 5: Generating the RDF Entities

The final two steps of the RDFGraphGen procedure is the actual generation of the synthetic RDF triples, which will comprise the synthetic RDF knowledge graph, and its serialization into an output file.

The synthetic RDF graphs are generated based on the shape maps which are the result of Step 3, as described in the previous section. The SHACL constraints [2] translated from the source SHACL document into the shape maps, describe precisely and in great detail how the generated RDF triples should look like:

- the *value type constraint components* define an ontology type for the generated entity;
- the *cardinality constraint components* denote the number of entities / values which should be generated;
- the *value range constraint components* and *string-based constraint components* describe the format, pattern and length of the values which should be generated;
- the *property pair constraint components* provide a way to constrain the value of one object based on the value of an object in another triple;
- the *logical constraint components* offer a choice of constraints to choose from while generating the synthetic data.

Below, we present Algorithm 2 which describes how a shape map created in Step 3 is used to generate synthetic RDF triples and recursively add them to the resulting synthetic RDF knowledge graph.

---

#### Algorithm 2 Generate RDF Entities based on a Shape Map

---

**Input** A Shape Map ( $SM$ ) and the data graph ( $G$ ) which is being generated

**MapToRDF**( $SM, G$ )

**Check for**  $SM\_properties$

    // Means that S contained nested property shapes

**If**  $SM\_properties$  exists:

**Create Node**

**For each**  $P_i : SM_i$  in  $SM\_properties$

**Add Node** -  $P_i$  - **MapToRDF**( $SM_i, G$ ) triple to  $G$

    return  $Node$

  // Means that S was a property shape that contained no other shapes, and only explained the property

**Else**

    return **GenerateObject**()

---

The process of generating synthetic RDF triples, presented in Algorithm 2, is a recursive algorithm comprised of the following main steps:

- First, all of the shape maps ( $SM_1 \dots SM_n$ ) based on node shapes which are not *pointed to* by another shape, are located.



- For each  $SM_i$  from the previous step, an unnamed node  $Node_i$  is generated and added to the result graph  $G$ .
- If  $SM_i$  contains properties, represented by the  $SM\_properties$  structure, a triple is added to the result graph  $G$  for each property-map pair  $P_j : SM_j$  in  $SM\_properties$ . In the triple,  $Node_i$  is the subject,  $P_j$  is the predicate and  $\mathbf{MapToRDF}(SM_j, G)$  is the object, i.e. the triple is structured as  $Node_i \rightarrow P_j \rightarrow \mathbf{MapToRDF}(SM_j, G)$ .  $\mathbf{MapToRDF}(SM_j, G)$  is the result node or literal generated recursively based on  $SM_j$ . If  $SM_i$  contains no  $SM\_properties$ , a literal is generated and added as an object in the triple.
- If  $NS_i$  points to another node shape  $NS_k$ , a number of entities  $E_k(1), \dots, E_k(f)$  are generated based on  $SM_k$ . The number is determined by the SHACL constraints for  $NS_k$ .

**Generating Values.** The generated synthetic object values in an RDF knowledge graph can be domain-specific or random, depending on whether the generated entity adheres to a known type from a well-known ontology, on the predicate used in the triple and on the SHACL constraints defined for the property in question.

In Algorithm 2, the process of generating an object value is replaced with the *GenerateObject()* function. Below, we will describe the process of generating an object’s value in a synthetic RDF triple.

When generating an object, 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 [11], 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 can pick a first name randomly from a set of pre-generated first names which it uses as a dictionary. The generator contains sets of pre-generated values for the most common properties from Schema.org, and for these properties values are randomly selected from these sets by the RDFGraphGen generator.

Even though the current version of the generator has an implementation of this functionality for a several entity types from the Schema.org vocabulary, it can easily be expanded to include more types from different ontologies, either general or domain-specific.

If the type of the entity for which a triple is being generated is unknown, the name of the predicate in the triple can provide useful information concerning the value of the object. For example, if the name of the predicate is `birthDate`, from a random or unknown ontology, the word `date` suggests that the object should be a date. Furthermore, if the predicate name contains the word `telephone` or `phone` and no data concerning the pattern is found in the input SHACL shape, a general pattern for a telephone number can be used by the generator to generate a random value which will make the generated synthetic RDF triple more relevant. The generator can easily be expanded to include more rules which are based on the names of previously unknown predicates it comes across.

When the value of the object cannot be picked from a pre-defined set of specific values, a function for generating a random, synthetic value is called. This function uses all of the constraints defined in the input SHACL shape which are related to the object, and which carry useful information about the object and its value. This function generates an object or a literal value based on a number of SHACL constraints: datatype, minimum length, maximum length, pattern, minimum and maximum value, etc.

This last part is what makes the RDFGraphGen generator general purpose and domain-independent: it can generate synthetic RDF triples and graphs from any domain, regardless of if it is explicitly familiar with it or not. The pre-generated sets of first names, last names, street names, professions, movie titles, book titles, etc., are solely *good to have* features, which increase the quality of the generated RDF knowledge graphs. However, that does not diminish the value of the synthetic graphs which the RDFGraphGen generator can generate when it works with a new and unfamiliar domain. The generality of RDFGraphGen is its most prominent feature.

**Outputting the Synthetic RDF Graph.** As a final step, the generator outputs the generated synthetic RDF graph into the output Turtle file. This file contains all synthetic RDF triples generated in the previous step.

As we mentioned before, the RDFGraphGen generator also uses the input parameter `entityCount`. This parameter determines the size of the generated synthetic RDF graph, which can be anything from a single entity, to a graph containing billions of RDF triples.

For the sake of simplicity, the number of entities to be generated relates to the first node graph contained in the input SHACL graph, in cases when multiple node graphs are present in it. The number of entities from the other node graphs will be determined by the relation between the first node graph and the other node graphs (e.g. similarly to the example presented in Section 3.3, where each person can have one or no addresses), or the generator will generate the same amount of entities from it as well, if it is not related to the first node graph.

### 3.3 Generated Example using “schema:Person”

In order to showcase how RDFGraphGen works for specific SHACL shapes as inputs, we present here one example of using the `schema:Person` class from Schema.org. Other examples, from various domains, using ontologies and types which are familiar or not to the generator, are presented in Appendix A. Even more examples are available on the project’s GitHub page [20].

In this example, the input SHACL shape refers to entities of the type `schema:Person`. From the definition, each such entity is constrained by SHACL to have: a given name and a last name, or a full name; exactly one date of birth, which will be earlier than a possible death date; a gender with one of the two available values; an email; a job title; a telephone number; an address, which has a specified complex type (it’s an object, not a literal). Additionally, the address objects have a street address which is a string value; a postal code which can be a string or an integer with values between 10.000 - 99.999.

The content of the input SHACL shape file is presented below:

---

```

input-shape-person.ttl
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix schema: <http://schema.org/> .
@prefix sh: <http://www.w3.org/ns/SHACL#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

schema:PersonShape
  a sh:NodeShape ;
  sh:targetClass schema:Person ;
  sh:xone (
    [
      sh:property [
        sh:path schema:givenName ;
        sh:datatype xsd:string ;
        sh:name "given name" ;
      ] ;
      sh:property [
        sh:path schema:familyName ;
        sh:datatype xsd:string ;
        sh:name "last name" ;
      ] ;
    ] ;
    [
      sh:path schema:name ;
      sh:datatype xsd:string ;
      sh:name "full name" ;
    ]
  ) ;
  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:property [
    sh:path schema:email ;
  ] ;
  sh:property [
    sh:path schema:jobTitle ;
  ] ;
  sh:property [
    sh:path schema:telephone ;
  ] ;
  sh:property [

```



```

        sh:path schema:address ;
        sh:node schema:AddressShape ;
    ] .

schema:AddressShape
  a sh:NodeShape ;
  sh:closed true ;
  sh:property [
    sh:path schema:streetAddress ;
    sh:datatype xsd:string ;
  ] ;
  sh:property [
    sh:path schema:postalCode ;
    sh:or (
      [
        sh:datatype xsd:string
      ]
      [
        sh:datatype xsd:integer
      ]
    ) ;
    sh:minInclusive 10000 ;
    sh:maxInclusive 99999 ;
  ] .

```

---

After running the RDFGraphGen generator on the input SHACL shapes file, and setting the parameter `entityCount` to 2, this is an example of a generated synthetic RDF graph:

---

```

output-graph-person.ttl
@prefix schema: <http://schema.org/> .
@prefix sh: <http://www.w3.org/ns/SHACL#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.org/ns#Node100> a schema:Person ;
  schema:address <http://example.org/ns#Node101> ;
  schema:birthDate "1955-07-07"^^xsd:date ;
  schema:deathDate "1981-07-07"^^xsd:date ;
  schema:email "ulysses_pate@gmail.com" ;
  schema:familyName "Pate" ;
  schema:gender "male" ;
  schema:givenName "Ulysses" ;
  schema:jobTitle "bartender" ;
  schema:telephone "647-466-552849" ;
  sh:description schema:PersonShape .

<http://example.org/ns#Node102> a schema:Person ;
  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 schema:PersonShape .

<http://example.org/ns#Node101> schema:postalCode 17481 ;
  schema:streetAddress "no. 3 Lily st" ;
  sh:description schema:AddressShape .

<http://example.org/ns#Node103> schema:postalCode "Fh6UpXLm" ;

```

```

schema:streetAddress "no. 1 Gillette ave" ;
sh:description schema:AddressShape .

```

---

Let us have a look at some of the noteworthy aspects of what happens when RDFGraphGen generates the synthetic RDF graph in this case. First, when it generates the name of each person entity, only one option from the `xone` constraint can be selected. In the first entity, the person has separate properties for the first name and the surname. In the second entity, the person has a single property for the full name, and the full name includes both the first name and the surname, as a single value.

Further, properties related to dates have a date value for the object, despite not containing a datatype constraint in the description. The generator extracts information from the property names `schema:birthDate` and `schema:deathDate`, and determines that the objects' values in the generated RDF triples should be dates.

The email values are comprised of the first name and the surname of the person. This has been specifically predefined in the generator, when working with `schema:Person` entities and their email addresses. Some level of randomness is still included with the delimiter between the first name and surname in the email address.

The generated phone numbers follow a specific pattern, because the generator recognizes that the value of the object should be a phone number. This is again based on the property name, 'schema:telephone' in this case, so the generator applies the predefined pattern constraint.

The address for each person is generated as a separate RDF entity, based on the SHACL shape in the input file, which constrains the address.

### 3.4 Packaging as a Python library

In order to make the RDFGraphGen generator more easily available for the community, we packaged it as a Python library and published it online. Now, any interested stakeholder can get it locally and use it as a command-line tool.

RDFGraphGen is available via the command: `pip install rdf-graph-gen`, which downloads and installs it locally on the user machine. Afterwards, RDFGraphGen can be used via the command-line, using the command:

```
rdfgen input-shape.ttl output-graph.ttl entity-count
```

The command `rdfgen` takes three arguments as input:

- The file containing the input SHACL graph (e.g. `input-shape.ttl`),
- The file where the generated synthetic RDF graph is to be written (e.g. `output-graph.ttl`), and
- The number of entities to be generated.

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 as a GitHub repository<sup>1</sup> [20].

Using a CI/CD pipeline based on GitHub actions, we release a new version of the RDFGraphGen generator on PyPi every time there is a change in the GitHub repository. The project is publicly available on PyPi<sup>2</sup> [21].

## 4 Discussion and Future Work

This is the first iteration and the first release of the RDFGraphGen generator. Even though it is general-purpose and can be used in any domain, we added a few additional rules for the most commonly used ontologies and their classes, in order to make a more user-friendly output of generated RDF triples, where the values are not completely random. This approach has its drawbacks: a question arises as to why were these specific ontologies and classes chosen over others, why was the list not expanded, etc. These are all valid questions, and the answer is that we had to stop somewhere, and release this first version.

In the future, we plan to address several shortcomings of RDFGraphGen. Currently, the interconnections among entities in the generated RDF graphs are limited, so our focus will be on producing more interconnected data graphs. Additionally, we aim to introduce an interface which will allow users to have a better control of the number of entities

---

<sup>1</sup>RDFGraphGen on GitHub: <https://github.com/mveco/RDFGraphGen>

<sup>2</sup>RDFGraphGen on PyPi: <https://pypi.org/project/rdf-graph-gen/>

generated for each node shape defined in the SHACL shapes graph, as well as enhance control over the connections between these generated entities.

The next step will involve thorough testing of the generator’s capabilities, followed by a comparison with other similar generators. Furthermore, adding support for multiple ontologies, their types, and properties is an ongoing task that we will continue to work on.

Additionally, in order to decentralize and democratize the work on RDFGraphGen, we have published it as open-source, so any interested stakeholder can either join in its collaborative development via GitHub, or fork it and make their own version of it.

## 5 Conclusion

In this paper, we introduce RDFGraphGen, a general-purpose, domain-agnostic synthetic RDF graphs generator, which is based on SHACL constraints. Even though generators for synthetic RDF data have been developed in the past, this is the first time a domain-independent generator is introduced, and it is the first time a generator is based on SHACL constraints.

Even though the SHACL standard has been developed for the purpose of validating existing RDF data, we reverse its role here and use it as a starting point to generate new, synthetic RDF data which are aligned with the constraints defined by the SHACL shapes.

The synthetic RDF knowledge graphs, generated by RDFGraphGen, can be used in many different scenarios in the software development cycle, in the domains of RDF data, Linked Data and the Semantic Web, such as: application testing, algorithm testing, application benchmarking, software quality control, training of machine learning models, etc.

The domain independence feature of RDFGraphGen is its main feature: the generated RDF graph will contain data from the domain which is described in the source SHACL shapes file. It can be data about people, movies, events, web pages, books, temperature measurements, scientific experiments, healthcare records, gene data, geographic locations, social media interactions, financial transactions, e-commerce products, biodiversity records, transportation schedules, educational courses, etc. Given the generality of SHACL and its approach, the same generality is available via the RDFGraphGen generator, as well.

In order to make the generated RDF graphs and the values in it more user-friendly, and closer to real-world values, the generator is able to recognize some of the most common classes and properties from the Schema.org ontology. Additionally, the generator can use the name of a given predicate to detect some of the more common datatypes, such as date, phone, email, etc. This reduces the amount of gibberish data, which is structurally valid from a SHACL point of view, but can be unwanted in some scenarios.

RDFGraphGen is very user-friendly: it can produce a large RDF knowledge graph with synthetic data, based on one or multiple SHACL shapes from a given domain, with just a single command-line instruction. The amount of RDF entities in the generated data can be controlled directly by the user.

The code of the RDFGraphGen generator is publicly available via its GitHub repository. It is also available as a public Python library, hosted on PyPi, making it available for use by any interested user via a command-line interface. This approach also allows us to further develop the generator and make the new changes transparent via these two public locations.

## 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 2023. <http://webdatacommons.org/structureddata/2023-12/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] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web: A New Form of Web Content that is Meaningful to Computers will Unleash a Revolution of New Possibilities. In *Linking the World’s Information: Essays on Tim Berners-Lee’s Invention of the World Wide Web*, pages 91–103. 2023.

- [6] Daniel Blum and Sara Cohen. Generating RDF for Application Testing. In *9th International Semantic Web Conference ISWC 2010*, page 105, 2010.
- [7] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 145–156, 2011.
- [8] 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.
- [9] 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.
- [10] Simon Gottschalk and Elena Demidova. Tab2KG: Semantic Table Interpretation With Lightweight Semantic Profiles. *Semantic Web*, 13(3):571–597, 2022.
- [11] 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.
- [12] Aidan Hogan. The Semantic Web: Two Decades On. *Semantic Web*, 11(1):169–185, 2020.
- [13] Milos Jovanovik and Mirko Spasić. Benchmarking Virtuoso 8 at the Mighty Storage Challenge 2018: Challenge Results. In *Semantic Web Challenges*, pages 24–35, 2018.
- [14] Milos Jovanovik and Mirko Spasić. Transforming Geospatial RDF Data into GeoSPARQL-Compliant Data: A Case of Traffic Data. In *Proceedings of the 16th International Conference on Informatics and Information Technologies*, pages 76–81, May 2019.
- [15] Milos Jovanovik, Timo Homburg, and Mirko Spasić. Software for the GeoSPARQL Compliance Benchmark. *Software Impacts*, 8:100071, 2021. doi: <https://doi.org/10.1016/j.simpa.2021.100071>.
- [16] Milos Jovanovik, Timo Homburg, and Mirko Spasić. A GeoSPARQL Compliance Benchmark. *ISPRS International Journal of Geo-Information*, 10(7), 2021. doi: 10.3390/ijgi10070487.
- [17] 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.
- [18] Mirko Spasić and Milos Jovanovik. MOCHA 2017 as a Challenge for Virtuoso. In *Semantic Web Challenges*, pages 21–32, 2017.
- [19] Mirko Spasić, Milos Jovanovik, and Arnau Prat-Pérez. An RDF Dataset Generator for the Social Network Benchmark with Real-World Coherence. In *Workshop on Benchmarking Linked Data (BLINK 2016), at the 15th International Semantic Web Conference (ISWC 2016)*, 2016.
- [20] Marija Vecovska and Milos Jovanovik. RDFGraphGen on GitHub, 2024. <https://github.com/mveco/RDFGraphGen>.
- [21] Marija Vecovska and Milos Jovanovik. RDFGraphGen on PyPi, 2024. <https://pypi.org/project/rdf-graph-gen/>.

## Appendix A More Generated Examples

In order to add more context and practically showcase the way the RDFGraphGen works, we will present several more examples of input SHACL shapes and generated synthetic RDF knowledge graphs based on them. Additional examples are available on the RDFGraphGen GitHub page [20].

### A.1 An Example of Using the “schema:Book” Class from Schema.org

This example uses the `schema:Book` class from the Schema.org vocabulary [11]. This class represents entities which are books, and the input SHACL shape file, listed below, contains constraints for each book entity, such as: each book can have an identifier which follows a specific pattern; each book can have a name which is a string; each book can have exactly one ISBN identifier; each book can have between 1 and 3 authors; etc. Additionally, the SHACL shape file contains a node shape for `schema:Author` entities as well, which represent the book authors. The SHACL constraints for the author entities specify that they each must have: a given name; a possible birth date and a death date; gender; an email address; etc.

---

```

input-shape-books.ttl
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix schema: <http://schema.org/> .
@prefix sh: <http://www.w3.org/ns/SHACL#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

schema:BookShape
  a sh:NodeShape ;
  sh:targetClass schema:Book ;
  sh:property [
    sh:path schema:identifier ;
    sh:maxCount 1 ;
    sh:pattern '^[a-z]{4}[0-9]{4}$' ;
  ] ;
  sh:property [
    sh:path schema:name ;
    sh:datatype xsd:string ;
  ] ;
  sh:property [
    sh:path schema:bookEdition ;
    sh:maxCount 1 ;
  ] ;
  sh:property [
    sh:path schema:isbn ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] ;
  sh:property [
    sh:path schema:numberOfPages ;
    sh:minInclusive 100;
  ] ;
  sh:property [
    sh:path schema:author;
    sh:node schema:AuthorShape ;
    sh:minCount 1 ;
    sh:maxCount 3 ;
  ] ;
  sh:property [
    sh:path schema:dateCreated ;
    sh:datatype xsd:date;
    sh:lessThan schema:datePublished ;
  ] ;
  sh:property [
    sh:path schema:datePublished ;
  ] ;
  sh:property [
    sh:path schema:genre ;

```

```

        sh:minCount 2 ;
        sh:maxCount 4 ;
        sh:description "Each book has to have at least 2 genres,
                        and a maximum of 4 genres." ;
    ] ;
    sh:property [
        sh:path schema:award ;
    ];
    sh:property [
        sh:path schema:inLanguage ;
        sh:in ( "en-USA" "en-UK" )
    ] .

schema:AuthorShape
    sh:targetClass schema:Person ;
    a sh:NodeShape ;
    sh:property [
        sh:path schema:givenName ;
        sh:datatype xsd:string ;
        sh:name "given name" ;
    ] ;
    sh:property [
        sh:path schema:birthDate ;
        sh:lessThan schema:deathDate ;
        sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path schema:gender ;
        sh:in ( "female" "male" ) ;
    ] ;
    sh:property [
        sh:path schema:email ;
    ] ;
    sh:property [
        sh:path schema:telephone ;
    ] .

```

After running RDFGraphGen on the input SHACL shapes file listed above, and setting the parameter entityCount to 1, below is an example of one generated synthetic RDF graph:

---

```

output-graph-books.ttl

```

---

```

@prefix schema: <http://schema.org/> .
@prefix sh: <http://www.w3.org/ns/SHACL#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.org/ns#Node100> a schema:Book ;
    schema:author <http://example.org/ns#Node101>,
        <http://example.org/ns#Node102> ;
    schema:award "The James Tait Black Memorial Prize" ;
    schema:bookEdition schema:EBook ;
    schema:dateCreated "1937-07-07"^^xsd:date ;
    schema:datePublished "1984-07-07"^^xsd:date ;
    schema:genre "Paranormal",
        "Soft Science Fiction",
        "Travel Writing" ;
    schema:identifier "wrzx5410" ;
    schema:inLanguage "en-UK" ;
    schema:isbn "602-5-30-585" ;
    schema:name "The Nature of Space and Time" ;
    schema:numberOfPages 121 ;
    sh:description schema:BookShape .

<http://example.org/ns#Node101> a schema:Person ;

```



```

    schema:birthDate "1946-07-07"^^xsd:date ;
    schema:deathDate "1982-07-07"^^xsd:date ;
    schema:email "cecile_848@gmail.com" ;
    schema:gender "female" ;
    schema:givenName "Cecile" ;
    schema:telephone "638539-768808" ;
    sh:description schema:AuthorShape .

<http://example.org/ns#Node102> a schema:Person ;
    schema:birthDate "1964-07-07"^^xsd:date ;
    schema:deathDate "1987-07-07"^^xsd:date ;
    schema:email "michel_807@gmail.com" ;
    schema:gender "male" ;
    schema:givenName "Michel" ;
    schema:telephone "050-097-6" ;
    sh:description schema:AuthorShape .

```

---

The SHACL constraints on the property `schema:identifier` define a pattern, and the generated value for the property in the entity corresponds to that pattern. This is achieved by generating a textual string based on the template using the `exrex` Python library.

The name of the book is randomly chosen from a predefined set of book titles, contained in the generator. The list can be modified and upgraded by users, whenever necessary, just like all the other predefined sets of values.

The property `schema:bookEdition` has a value from the appropriate enumeration for that property, as defined in the Schema.org vocabulary.

The property `schema:genre` has a minimum (2) and a maximum (4) count constraint, so the generator has generated 3 genres for the synthetic entity. The book genres are also selected randomly from a predefined list, which comes with the generator.

The author property points to another SHACL shape and has a minimum (1) and a maximum (3) count constraint. So, in this case, two author entities are generated and added as values for that property. The two author entities are then also fully generated by `RDFGraphGen`, based on the SHACL constraints.

## A.2 An Example of Using the “`schema:TVSeries`” Class from Schema.org

Here we have an example which uses the `schema:TVSeries` class from the Schema.org vocabulary [11]. This class represents entities for TV series. The SHACL shape file which defines its constraints, contains definitions of the following about each TV series entity: it can have a director; it can have a minimum of 3 actors; it can have numerous seasons and episodes, but the number of seasons should be lower than the number of episodes; it can have a title EIDR which follows a specific pattern; etc. Additionally, the SHACL shape file listed below contains definitions of SHACL constraints for the directors and actors of the TV series. They are both entities from the `schema:Person` class from Schema.org, and they have the usual property constraints which we already saw in the previous examples.

---

```

input-shape-tvseries.ttl
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix schema: <http://schema.org/> .
@prefix sh: <http://www.w3.org/ns/SHACL#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

schema:TVSeriesShape
  a sh:NodeShape ;
  sh:targetClass schema:TVSeries;
  sh:property [
    sh:path schema:director;
    sh:node schema:DirectorShape ;
  ] ;
  sh:property [
    sh:path schema:actor;
    sh:node schema:ActorShape ;
    sh:minCount 3 ;
  ] ;

```

```

] ;
sh:property [
  sh:path schema:season ;
  sh:datatype xsd:integer ;
  sh:equals schema:numberOfSeasons ;
  sh:minInclusive 0 ;
] ;
sh:property [
  sh:path schema:numberOfEpisodes ;
  sh:minInclusive 0 ;
] ;
sh:property [
  sh:path schema:numberOfSeasons ;
  sh:lessThan schema:numberOfEpisodes ;
  sh:minInclusive 0 ;
] ;
sh:property [
  sh:path schema:titleEIDR ;
  sh:datatype xsd:string ;
  sh:pattern '[A-Z1-9.-/]{10,20}';
] ;
sh:property [
  sh:path schema:name ;
  sh:datatype xsd:string ;
] ;
sh:property [
  sh:path schema:startDate ;
  sh:datatype xsd:date;
  sh:lessThan schema:datePublished ;
] ;
sh:property [
  sh:path schema:endDate ;
  sh:datatype xsd:date;
] ;
sh:property [
  sh:path schema:datePublished ;
  sh:lessThanOrEquals schema:endDate ;
] ;
sh:property [
  sh:path schema:genre ;
] .

schema:DirectorShape
  sh:targetClass schema:Person ;
  a sh:NodeShape ;
  sh:property [
    sh:path schema:givenName ;
    sh:datatype xsd:string ;
  ] ;
  sh:property [
    sh:path schema:familyName ;
    sh:datatype xsd:string ;
  ] ;
  sh:property [
    sh:path schema:gender ;
    sh:in ( "female" "male" ) ;
  ] ;
  sh:property [
    sh:path schema:email ;
  ] ;
  sh:property [
    sh:path schema:telephone ;
  ] .

schema:ActorShape

```

```

sh:targetClass schema:Person ;
a sh:NodeShape ;
sh:property [
  sh:path schema:name ;
  sh:datatype xsd:string ;
  sh:name "given name" ;
] ;
sh:property [
  sh:path schema:gender ;
  sh:in ( "female" "male" ) ;
] .

```

---

Below is the content of one synthetic RDF graph, generated by RDFGraphGen with entityCount set to 1.

---

output-graph-tvseries.ttl

---

```

@prefix schema: <http://schema.org/> .
@prefix sh: <http://www.w3.org/ns/SHACL#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.org/ns#Node100> a schema:TVSeries ;
  schema:actor <http://example.org/ns#Node102>,
    <http://example.org/ns#Node103>,
    <http://example.org/ns#Node104> ;
  schema:datePublished "1967-07-07"^^xsd:date ;
  schema:director <http://example.org/ns#Node101> ;
  schema:endDate "1986-07-07"^^xsd:date ;
  schema:genre "Supernatural Thriller" ;
  schema:name "Life After Beth" ;
  schema:numberOfEpisodes 31 ;
  schema:numberOfSeasons 4 ;
  schema:season 4 ;
  schema:startDate "1917-07-07"^^xsd:date ;
  schema:titleEIDR "7ZFCQWFG32." ;
  sh:description schema:TVSeriesShape .

<http://example.org/ns#Node101> a schema:Person ;
  schema:email "denna_sosa@gmail.com" ;
  schema:familyName "Sosa" ;
  schema:gender "female" ;
  schema:givenName "Denna" ;
  schema:telephone "921-682-3863852" ;
  sh:description schema:DirectorShape .

<http://example.org/ns#Node102> a schema:Person ;
  schema:gender "female" ;
  schema:name "Debi Snow" ;
  sh:description schema:ActorShape .

<http://example.org/ns#Node103> a schema:Person ;
  schema:gender "male" ;
  schema:name "Martainn Foster" ;
  sh:description schema:ActorShape .

<http://example.org/ns#Node104> a schema:Person ;
  schema:gender "male" ;
  schema:name "Waine Mcclain" ;
  sh:description schema:ActorShape .

```

---

In this example, the TV series SHACL shape contains two properties that point to other SHACL shapes: one to a director entity, and one to actor entities. The director entity constraint does not have a minimum or maximum cardinality constraint, so a single director entity is generated for each TV series. On the other hand, the actor entity

constraint has a minimum cardinality of 3, so at least 3 actor entities are generated and placed in the corresponding semantic relation to the TV series entity.

The number of seasons is constrained to be less than the number of episodes, and the generator abides to this.

The `schema:season` property is constrained in the input SHACL file to have the same value as the `schema:numberOfSeasons` property, and this is followed by the generator.

Additionally, the `schema:startDate` property is defined to have a date value which is less then the date value of `schema:datePublished` property, while `schema:datePublished` should be less or equal to `schema:endDate`. These constraints are again respected by RDFGraphGen, as can be seen from the generated RDF graph.

### A.3 An Example of Using Classes and Properties from an Unknown Ontology

The examples so far involved well-known entity types from Schema.org. As we explained before, the generator has several predefined sets of possible values for the more common Schema.org classes and properties, for usability sake. However, the main feature of RDFGraphGen is its universality and domain independence. In order to show that it can work with other ontologies, which are not well-known for the generator, we will use an example here which uses the non-existent example ontology, with it's classes and properties.

---

```

input-shape.ttl

@prefix schema: <http://schema.org/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.com/ns#> .

schema:ExampleShape
  a sh:NodeShape ;
  sh:targetClass ex:Product;
  sh:property [
    sh:path ex:name ;
    sh:datatype xsd:string;
    sh:minLength 10;
    sh:maxLength 10;
  ] ;
  sh:property [
    sh:path ex:identifier ;
    sh:minCount 3;
    sh:maxCount 6;
  ];
  sh:property [
    sh:path ex:dateOfProduction ;
    sh:lessThan ex:dateOfExpiration ;
  ] ;
  sh:property [
    sh:path ex:dateOfExpiration ;
    sh:minInclusive "2007-02-10"^^xsd:date ;
    sh:maxInclusive "2007-05-10"^^xsd:date ;
  ] .

```

---

When we use the RDFGraphGen generator on the input file, with the `entityCount` value set to 2, we get a synthetic RDF graph which follows the SHACL constraints, and looks like the sample below.

---

```

output-graph.ttl

@prefix ex: <http://example.com/ns#> .
@prefix schema: <http://schema.org/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.org/ns#Node100> a ex:Product ;
  ex:identifier "I6sTsyxw",
    "nlctTaNsK5g",

```

---

```

        "pbuXXfsY",
        "r03mdpRk" ;
    ex:dateOfProduction "1957-04-10"^^xsd:date ;
    ex:dateOfExpiration "2007-04-10"^^xsd:date ;
    ex:name "1bCJ3ZXgGK" ;
    sh:description schema:ExampleShape .

<http://example.org/ns#Node101> a ex:Product ;
    ex:identifier "AArBAfHB",
        "Q1J4qXYuag",
        "YQJGquoIHx4" ;
    ex:dateOfProduction "1984-03-10"^^xsd:date ;
    ex:dateOfExpiration "2007-03-10"^^xsd:date ;
    ex:name "H2mk10GQDl" ;
    sh:description schema:ExampleShape .

```

---

The `ex:name` property, even though it is not known for the generator, has a specified datatype in the SHACL file (`xsd:string`), and a maximum and minimum length. This helps the generator generate a random value for each synthetic RDF triple which uses this property as a predicate.

The `ex:identifier` property has no defined datatype, so the generator assumes string values. Given the cardinality constraints of 3 and 6, the `RDFGraphGen` generates random string values for the synthetic RDF triples it generates with this property.

The `ex:dateOfProduction` property does not have a datatype, but has a SHACL constraint relating it to another property, which has `xsd:date` as minimum and maximum values. This leads the generator to assume that both properties have dates as values. The SHACL constraints define that `ex:dateOfProduction` has to have a lesser value than the `ex:dateOfExpiration` property, and additionally that the `ex:dateOfExpiration` property has a value in the defined range. Following these SHACL constraints, the `RDFGraphGen` generates the output shown above.