RDFMutate: Mutation-Based Generation of Knowledge Graphs

Tobias John[1], Einar Broch Johnsen[1], and Eduard Kamburjan[2,1]

[1] University of Oslo, Oslo, Norway
{tobiajoh,einarj}@ifi.uio.no
[2] IT University of Copenhagen, Copenhagen, Denmark
eduard.kamburjan@itu.dk

Abstract. This paper introduces RDFMutate, the first mutation-based tool to generate RDF knowledge graphs. RDFMutate enables developers to analyze the robustness of applications that operate on RDF, by generating variants of seed knowledge graphs, which are mutated according to a set of mutation operations and SHACL constraints. In contrast to existing tools to generate synthetic RDF graphs, RDFMutate provides a mutation-based approach that is accessible for both researchers and application developers, by providing a framework to define mutation rules and flexible selection of generated graphs.

Keywords: Graph Generator · Graph Mutation · Application Testing

1 Introduction

Software applications that work with knowledge graphs (KGs) are subject to multiple quality criteria. While the quality of the graph data is a crucial concern, the quality of the software itself is of equal importance. Software testing of such applications relies on readily available input graphs, and automated software testing relies on readily available graph generators. As the input graphs must satisfy numerous constraints to be considered valid input, e.g., use the right vocabulary, be logically consistent or satisfy SHACL constraints, the use of arbitrary graphs is not possible. Furthermore, a few manually created KGs are often available for an application.

Synthetic KGs can be used as test inputs to perform integration and system testing of applications [14, 20] and have lead to the discovery of several, previously unknown bugs in state-of-the-art Web Ontology Language (OWL) reasoners and other mature software tools [16, 20]. Two approaches to generate synthetic KGs have been used for testing applications: language-based methods [20] and mutation-based methods [14]. While there is decent tool support for language-based generation of KGs [36], we are not aware of existing tools for mutation-based generation of KGs. Our tool, RDFMutate, fills this gap. It is available on github[3] together with its documentation[4].

[3] https://github.com/smolang/RDFMutate
[4] https://github.com/smolang/RDFMutate/wiki

RDFMutate generates Resource Description Framework (RDF) graphs [24] based on mutations. Given an existing KG, called *seed*, it applies a sequence of mutation operators to generate new graphs. For example, it may remove or add new nodes. The final KG is the result of a sequence of mutations. To ensure that it is a valid input, RDFMutate enables the following customization options.

Mutation Operators RDFMutate predefines general, application-independent mutation operators, and allows the user to input additional mutation operators. These operators can be provided either as Java code, or in a format based on SWRL [11] rules.

Shape Validation RDFMutate applies SHACL shapes to ensure the structure the result. If the resulting graph does not adhere to the shapes, different mutation operators are applied until an eventual adherence. In particular, users can reuse SHACL shapes that already exist for specific domains.

In the end, not the whole generated KG is synthetic, but only local changes to parts of the seed KG are applied. If the mutation operators are chosen correctly, with respect to the application domain, the generated mutant KGs are still within the domain that is of interest.

This paper contains an evaluation of RDFMutate in two dimensions: First, we demonstrate how RDFMutate can be used to find unknown bugs in common applications working on KGs in a *qualitative evaluation*. Secondly, we analyze the *performance* of RDFMutate and show that the generation times are short enough to use RDFMutate in practice. Our main contribution are as follows:

- The first tool for mutation-based generation of RDF graphs.
- Custom mutation operators that are application-specific. The mutation operators can be specified using a custom input format based on SWRL rules.
- Synthetic KG generation with respect to provided constraints. The constraints can contain SHACL shapes that the KG must conform to.

Predecessors of RDFMutate have been applied in previous work [14, 16]. RDFMutate was developed with a focus on reuse and usability. Changes are, beyond minor refactoring, the following: (a) A configurable frontend to define all aspects of the mutation process. (b) Custom mutation operators in the form of SWRL rules are supported. (c) Custom strategies to select mutation operators. Furthermore, we have provided extensive online documentation that describes how to install, use and extend RDFMutate. This paper also provides a new quantitative performance evaluation

This work is structured as follows: We discuss related works in Section 2 and the architecture of RDFMutate in Section 3. Usage and evaluation are given in Section 4. Section 5 discusses use cases, possible extensions, and limitations.

2 Related Work

RDF generation tools. There are numerous tools to generate synthetic RDF graphs, and we only discuss those that consider additional constraints on the

Table 1: Comparison of KG generation tool features. **#Schema Properties** refers to the number of IRIs that can be generated to express schema information.

Tool	**Generation Method**	**#Schema properties**	**Constraints**	**Tool available**
GDD [4]	schema-based	all	GDD^x conformance	no
DLCC [32]	random	3	no	yes
PyGraft [13]	schema-based	13	consistency check	yes
RDFGraphGen [35]	schema-based	all	SHACL conformance	yes
RDFmutate	mutation-based	all	consistency check + SHACL conformance	yes

resulting KG, and are not merely generating triples representing data. Tools that generate RDF graphs but do not fulfill this criterion are discussed in [13].

Tools that generate graphs according to constraints follow one of two approaches: (i) Consistency of the encoded axioms, or (ii) the conformance to a schema such as SHACL or GDD^x. However, our tool is the only one that takes both forms of constraints into account.

Table 1 shows an overview of methods that are comparable to our tool. The tools GDD, DLCC, PyGraft and RDFGraphGen all use some schema information, e.g. SHACL shapes, GDDs or OWL axioms to generate the KG. Our tool is the only approach that is based on mutation operators to generate the KG.

Some of them, DLCC and PyGraft, are only able to generate specific schema properties, i.e. properties that express subclass relations or other OWL axioms, while our tool is able to generate arbitrary schema information and use a reasoner to treat it accordingly in the consistency check. Additionally, our tool is able to generate not only object-property relations but also data-property relations with literals, which PyGraft and DLCC can not do. In general, our tool RDFMutate can generate all kinds of RDF graphs, which not all other tools can do.

Ontology Mutation. There exist some proposals for mutation operators on OWL ontologies for quality control, similar to methods of test-driven development of ontologies [23]. The approaches focus on different aspects, e.g. on domain-specific mutation operators [25], on quality control of ontologies [1] or on proposing low-level mutation operators [31]. Tool support for these approaches is limited. Only the alpha version of a tool for the mutations discussed in [1] is available, which lacks documentation and does not seem to be maintained any more. Additionally, our tool is not restricted to a specific set of operations on OWL ontologies but allows using arbitrary mutation operations on the triples in the KG.

Test Case Generators. Test case generators, which are also known as *fuzzers* [36], are well established tools in testing software applications [28]. The idea is to generate a huge number of examples of random input data for the application that is tested and monitor its execution for undesired behavior. There are many such tools available [2, 5, 8, 9, 33], which use different techniques techniques, such as language-based generation or mutation-based generation. While language-based generation bears similarity to schema-based approaches to RDF generation, our tool falls in the category of mutation-based generators, which have so far not been explored for KGs. It is the only tool that is specifically designed to generated RDF graphs. In particular, the mutations that we use are not defined on a *syntactic* level, i.e, on the input format, which varies for different serializations of RDF graphs. Instead, we use mutation operators that are defined on the *semantic* level, i.e. the interpretation of input data as a graph structure.

3 Architecture and Description

RDFMutate generates knowledge graphs (KGs) in the form of RDF graphs [24]. The way how the KGs are generated is defined by mutation operators and masks.

Mutation Operator. RDFMutate uses *mutation operators*, which describe changes in KGs. Most mutation operators contain a graph pattern to select the locations where they can be applied and a graph pattern how the KG should be changed. Those patterns contain variables, i.e., a mutation operator can be applicable at multiple locations in a KG. One can distinguish two types of mutation operators based on their semantics: (i) domain-independent mutation operators represent general mutations that can be used across diverse domains and (ii) domain-specific mutation operators are designed for KGs from one particular domain [14]. Section 3.3 describes how to select and define mutation operators.

Mask. Masks are a concept to define a criterion when a generated KG is valid and were introduced in [14]. A mask can have two components: (i) a set of SHACL shapes and (ii) a reasoner, e.g. an OWL reasoner, that decides if the KG is consistent, e.g. if the entailed OWL axioms form a consistent set. A generated KG is *valid*, iff it conforms to the SHACL shapes and is classified "consistent" by the reasoner. Both components, the shapes and the reasoner, can be omitted to only use one of those criteria. It is also possible to use no criterion at all and we call such a mask an *empty mask*.

3.1 System Overview

Figure 1 shows the architecture of RDFMutate, which at a high level consists of a *frontend* managing the input, and a *backend* that manages the actual generation.

Frontend. The frontend of RDFMutate parses all the input files and saves the generated mutant KGs. The frontend also checks if all the input elements are correct, e.g., it checks if the output file already exists and no generation is triggered if the mutant KG can not be saved. More details about how the mutations and shapes can be specified can be found in Section 3.3 and Section 3.4. Note, that all the files shown in Figure 1 contain RDF graphs, i.e., we use a format that users are already familiar with.

Additionally, we use a yaml-configuration file to contain all the information about how to perform the mutation and where to find all resources. The frontend

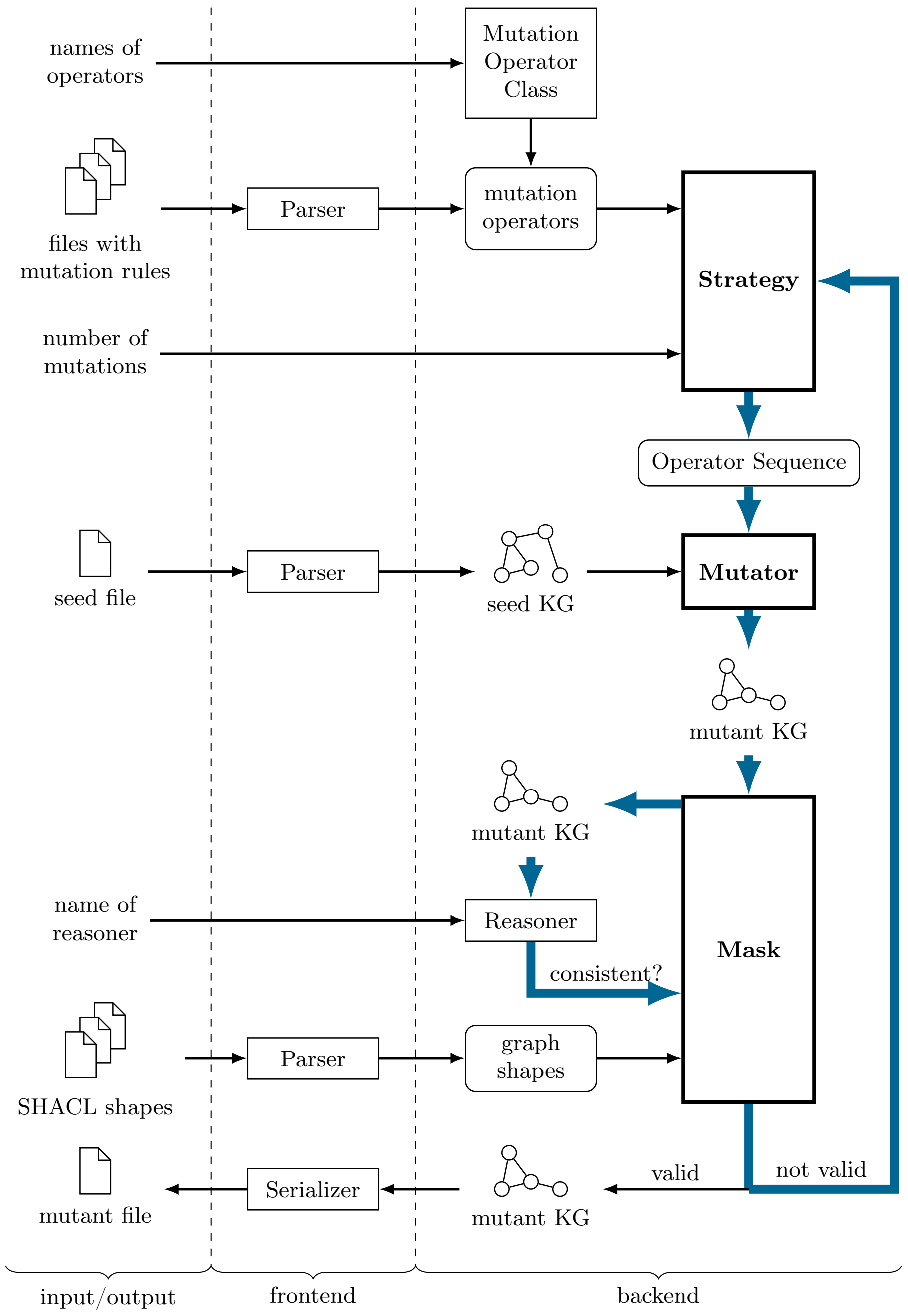


Fig. 1: The most important components of RDFMutate's architecture. The bold (blue) arrows are used iteratively (starting from the strategy) until a valid mutant is found. All the elements on the left are specified via the configuration file.

```yaml
seed_graph:
    file: examples/simpleKG.ttl
    type: rdf
output_graph:
    file: examples/outputKG.ttl
    overwrite: true
    type: rdf
number_of_mutations: 10
number_of_mutants: 5
strategy:
    name: random
mutation_operators:
    - module:
        location: org.smolang.robust.mutant.operators
        operators:
            - className: AddSubclassRelationMutation
            - className: AddObjectPropertyRelationMutation
    - resource:
        file: examples/addRelation.ttl
        syntax: swrl
condition:
    reasoning:
        consistency: true
        reasoner: hermit
    masks:
        - file: examples/AsubClassOfB.ttl
        - file: examples/AsubClassOfC.ttl
```

Fig. 2: An example yaml-configuration file that contains all information on how to run the generation of the mutant KGs.

also parses this file (omitted from Figure 1 for clarity). An example of such a configuration file is depicted in Figure 2. An explanation of all the elements that can be specified in the configuration file can be found online [19].

Backend. The backend is the main component of RDFMutate. It is depicted on the right-hand side in Figure 1. We describe the most important components and the Kotlin classes that implement them.

RDFMutate contains a class that represents mutation operators. They implement a function that takes a KG and applies a change to it, i.e., performs a mutation. Currently, RDFMutate provides 59 different mutation operators. Descriptions of their behaviors are contained in our online documentation [19]. The following three components implement the workflow of RDFMutate.

Strategy Provided a set of mutation operators and an integer, the *search strategy* produces a sequence of mutation operators of the requested length. RDFMutate comes with a default strategy that selects mutation operators randomly from the set of mutation operators. To avoid generating the same sequence again, a strategy can also hold some state information.

Mutator The *mutator* takes a sequence of mutation operators and applies them to a seed KG one after the other. If an operator can be applied at several locations in the KG, one such location is chosen randomly. At the end, a mutant KG is obtained.

Mask The *mask* decides the validity of a mutant KG, i.e, its adherence to external constraint. The mask might consult a reasoner to check the consistency of the KG and RDFMutate provides HermiT [7], Pellet [34], ELK [22] and the Apache-Jena reasoner as options. Furthermore, the mask checks, if a mutant KG conforms to provided SHACL shapes.

Execution. The generation of a valid mutant KG works a follows:

1. RDFMutate reads the yaml-configuration file that contains the information about how to perform the generation (this step is omitted in Figure 1 for the sake of clarity). All files mentioned in the configuration file are parsed.
2. The set of selected mutation operators is created handed over to the strategy component. The set can contain existing operators or operators specified by the user via files containing mutation rules.
3. The strategy component generates a sequences of mutation operators.
4. The generated sequence of mutation operators is handed to the mutator, which applies is to the seed KG to generate the mutant KG.
5. The mask analyzes if the mutant KG is valid by taking the provided SHACL shapes and the consistency classification from the reasoner into account.
6. If the mutant KG is not valid, the execution continues at step 3. If the mutant KG is valid, it is saved to the output file using the desired serialization.

Using the yaml-configuration, the user can also select to generate a batch of several mutant KGs. In this case, the cycle of generating mutant KGs is repeated until the desired number of KGs is generated.

3.2 Implementation

RDFMutate is implemented in Kotlin and uses Apache Jena 5.2 [6] to parse and serialize the files containing RDF graphs. For the consistency checks, off-the-shelf reasoners are used, namely HermiT [7], Pellet [34] and ELK [22].

3.3 Mutation Specification

There are two ways to specify the mutation operators that RDFMutate should consider to apply: (i) name classes that implement operators and (ii) import operators from files. Both options are shown in the configuration file in Figure 2.

For the first option, the name of the module that contains the operator and the name of the class that implements the operator are provided. The class is

```
class PizzaMutation(model: Model) : Mutation(model) {
  val topping: Resource = model.getResource(":Topping")
  val pizza: Resource = model.getResource(":Pizza")
  val hasTopping: Property = model.getProperty(":hasTopping")

  override fun createMutation() {
    val toppings = model.listResourcesWithProperty(RDF.type, topping)
    val t = toppings.toSet().random()
    val p = model.createResource(":newPizza" + Random.nextInt())
    addSet.add(model.createStatement(p, RDF.type, pizza))
    addSet.add(model.createStatement(p, hasTopping, t))
    super.createMutation()
  }
}
```

(a) Example as a Kotlin class (import omitted).

$\texttt{rdfmutate:newNode(?p)} \land \texttt{:Topping(?t)} \rightarrow \texttt{:Pizza(?p)} \land \texttt{:hasTopping(?p, ?t)}$

(b) Example as a SWRL rule (`?p` and `?t` are IRIs of SWRL variables).

Fig. 3: Comparison of two versions of the same mutation operator.

dynamically loaded at runtime. The user can refer to multiple operators from the same module and several modules can be listed to import operators from.

The second option is more flexible and allows to import mutation operators from files. We chose to encode the mutation operators as RDF graphs, more specifically, we use SWRL rules [11]. SWRL is a rule language for the semantic web that describes rules of the form body→head, which are implications between two sets of assertions about graph structures. We use the syntax of the SWRL rules, but use a custom interpretation. While the standard interpretation of a SWRL rule is to infer additional assertions in the KG, we interpret a SWRL rule as a *change* of the KG, which is similar to the interpretation in [26]. A mutation operator described by a SWRL rule can be applied at all locations in a KG where its body matches. Note, that this matching is done directly on the KG, without prior inference steps. When the mutation operator is applied, the consequences described by the assertions in its head are performed. In the simplest case, this means that the atoms in the head are added to the KG. To express more complex consequences, we use SWRL's feature of using custom "buit-in" atoms. A list of all the built-ins that RDFMutate can parse and execute is shown in Table 2.

As an additional extension, we allow arbitrary IRIs to be used as properties in property-atoms, whereas the SWRL standard only allows object or data properties. This extension allows the user to target arbitrary triples, including those that specify schema-information.

Example 1. Consider the mutation operator that is shown in Figure 3. The operator selects a node `t` that is of type `:Topping` and creates a new node of type `:Pizza` that has `t` as a topping. I.e., the operator selects a random topping and creates a new pizza with this topping.

Table 2: Additional built-ins that we use to specify mutation operators. We use the prefix `owl:` for `http://www.w3.org/2002/07/owl#` and `rdfmutate:` for `https://smolang.org/rdfMutate#`. #**arg** is the number of arguments.

IRI	**#arg**	**place**	**semantic**
`owl:NegativePropertyAssertion`	3	body	relation is not contained in KG
`rdfmutate:newNode`	1	body	argument (must be a variable) is a new node that will be added to the KG
`owl:NegativePropertyAssertion`	3	head	relation is removed from KG
`rdfmutate:deleteNode`	1	head	all triples containing this node are deleted from KG
`rdfmutate:replaceWith`	2	head	the first node is replaced by the second node in all triples

On the top of Figure 3, the operator is implemented as a Kotlin class. The KG is represented as the Jena Model "model". To make the additions to the KG, the new statements, i.e. triples, are added to the attribute "addSet". The actual performance of the addition is taken care of by the super-class "Mutation".

On the bottom of Figure 3, the same operator is expressed using a SWRL rule. Apart from declaring the nodes with the IRIs `?p` and `?t` as SWRL variables, no further elements need to be specified.

3.4 Mask Specification

To specify a mask, two types of constraints can be specified in the configuration file (see Figure 2): (i) the type of reasoning and (ii) the mask shapes. If the generated KG should be consistent, the corresponding flag needs to be set and the name of a reasoner needs to be provided. Furthermore, a list of files with SHACL shapes can be provided. All the shapes from all the files are combined into one shape graph by RDFMutate and the mutant KG is only valid if it conforms to this combined shape graph.

4 Usage and Evaluation

To demonstrate how to use RDFMutate, we first describe the general procedure followed by discussing RDFMutate's capabilities using two case studies where KGs generated with RDFMutate were used to analyze software applications. Afterwards, we evaluate the performance of RDFMutate. In particular, we investigate how much time and attempts are needed to generate (valid) KGs.

4.1 Usage

To use RDFMutate, one has to provide all the necessary inputs, i.e. the elements on the left in Figure 1, and provide them via a configuration file (see Figure 2). In particular, one has to perform the following steps.

Setup. The user selects a set of suitable seed KGs, possibly from existing or recorded inputs to the systems. Afterwards, mutation operators need to be selected, which should keep a KG within the domain when they are applied. If such operators are not provided by RDFMutate, they must be specified using SWRL rules or developed manually. Constraints that are not ensured by the mutation operators but are relevant for the domain are encoded using SHACL shapes.

Constraint and operator selection can be iterative [14], i.e., if KGs are generated that intuitively should not be considered valid, then new constraints or operators may be added.

Configuration. The number of mutation operators that is applied needs to be selected. Because the optimal number depends on the domain and is hard to predict, we advice to start with a low number and increase it, if the generated KGs are too similar to seed KG. One also needs to decided if the KG must be consistent and if yes, a reasoner must be selected. The number of generated KGs needs to be defined. We advice to start with a mall number and observe if the generated KGs are shaped as expected.

All this information is put in the configuration file (see Figure 2).

Execution. One runs RDFMutate using the command line as follows, assuming the configuration is in the file `configuration.yaml`:

```
java -jar rdfmutate.jar -config=configuration.yaml
```

4.2 Qualitative Evaluation

To demonstrate the capabilitis of RDFMutate, we briefly report on two instances where preliminary versions of RDFMutate have been used to test applications working with knowledge graphs. Detailed discussions of the setup and results can be found in the corresponding publications [14, 16].

Testing OWL Reasoners. RDFMutate has been successfully used to perform system testing on OWL reasoners [16]. More specifically, common reasoners for the OWL-EL profile where tested. To test the reasoners, RDFMutate was set up to create KGs that contain ontologies within the restrictive OWL-EL profile [12]. Ontologies from the 2015 OWL Reasoner Evaluation competition [30] where used as seed KGs. Using RDFMutate and 55 well-chosen mutation operators, the generated mutant KGs differ from the seed KGs but only contain ontologies within the OWL-EL profile. At the same time, the mutation operators ensure that all features, i.e. types of axioms, that are allowed in the OWL-EL profile occur in some of the generated KGs.

This testing campaign lead to the discovery of six previously unknown bugs, five bugs in the reasoner Pellet and one bug in the reasoner HermiT [15]. Table 3 contains an overview of the found bugs. Most of the bugs are exceptions on valid inputs, some of them are logical errors where the class hierarchy computed by the reasoners is not correct. This shows that RDFMutate can be used to find crucial bugs in common applications working with KGs.

Table 3: Bugs when testing OWL-EL reasoners. The issueId refers to the id in the corresponding online issue tracker [29] (this table is copied from [16])

reasoner	bugID	issueId	type	summary
Pellet	P1	94	Exception	Exception when doing pre-computation for class hierarchy; occurs non-deterministically
	P2	93	Completeness	A sub-class axiom is missing from inferred class hierarchy; combination of reflexive property and existential quantification
	P3	97	Exception	Exception when doing pre-computation for class hierarchy
	P4	95	Exception	Exception when doing pre-computation for class hierarchy
	P5	96	Completeness	A sub-class axiom is missing from inferred class hierarchy because sub-typing of different string data types is not considered
HermiT	H1	—	Exception	Exception when reasoner is initiated if ontology contains the axiom $\bot \sqsubseteq \top$

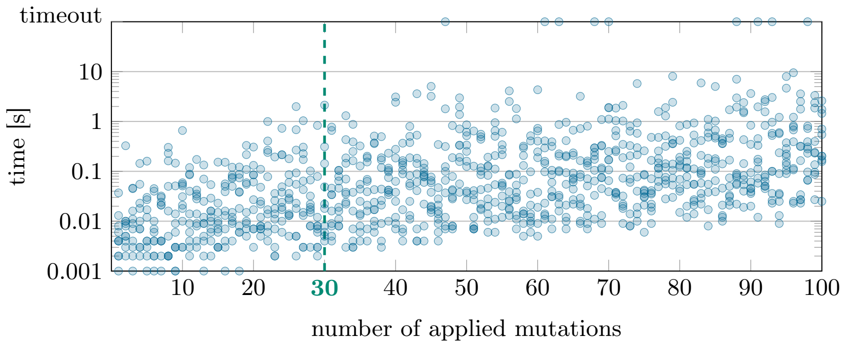
Extracting Shape Constraints for Applications interacting with KGs. RDFMutate has been successfully used to perform integration testing of software applications that use a KG as a knowledge base [14]. The tested applications have strong constraints on the shape of the KGs as they need to extract specific information from the KGs in order to work correctly. Our goal was to extract those constraints in the form of SHACL shapes. We did so by iteratively refining the SHACL shapes that the generated KGs need to conform to until no KGs could be generated where the applications did not work correctly. The refinement was performed manually by domain experts based on the results of the test runs. We used domain-specific and domain-independent mutation operators for this study and were able to extract shapes for all tested applications.

4.3 Performance Evaluation

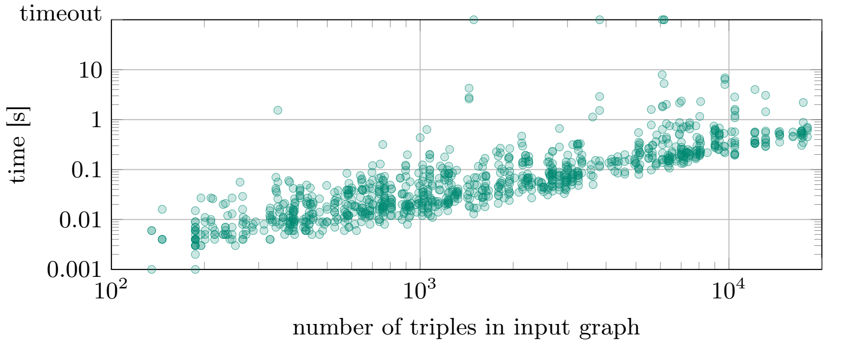
We evaluate the performance of RDFMutate in two dimensions: (i) the time to generate a mutant KG and (ii) the number of attempts that are necessary to generate a valid mutant in the presence of restrictive masks.

The variability in specifying mutation operators, input KGs and masks makes it non-trivial to evaluate RDFMutate in a way that resembles how users will actually use it. To consider reasonable generation scenarios, we base our evaluation on the existing use cases, where the selected mutation operators and masks have been able to generate helpful mutant KGs.

Scripts and all data to replicate the evaluation are available on Zenodo [17]. We ran the experiments on a laptop with an Intel i7-1165G7 CPU @ 2.80GHz running Ubuntu 22.04 with 8GB RAM, which took about 85 minutes.



(a) Generation time w.r.t. number of mutations that are applied to input.



(b) Generation time w.r.t. size of input graph when 30 mutations are applied.

Fig. 4: Time to generate. For both plots, we generate 1000 data points, used a timeout of 10s and recorded computation time in milliseconds.

Execution Time. RDFMutate needs to be fast enough to generate mutant KGs in reasonable time. Ideally, the generation time should be at most a few seconds.

We use the campaign of testing OWL-EL reasoners as the basis for this evaluation. We chose this setup as it was the only existing one with a variety of input KGs and because generation time is very relevant for this case as the run time of the reasoners is rather short. We use the same setup as the original study [16]: the same 307 input KGs, the same 55 mutation operators and an empty mask, representing no restrictions on the shape of the generated KGs.

We evaluate the generation time in two experiments, based on two parameters: (i) the number of applied mutation operators and (ii) the size of the input KG. In the original campaign, 30 mutation operators where applied per mutant KG. Hence, we consider applying 1–100 mutation operators in the first experiment and applying 30 mutation operators in the second experiment. We sample 1000 data points per experiment, selecting the input KGs randomly. We use a time limit of 10s for the generation.

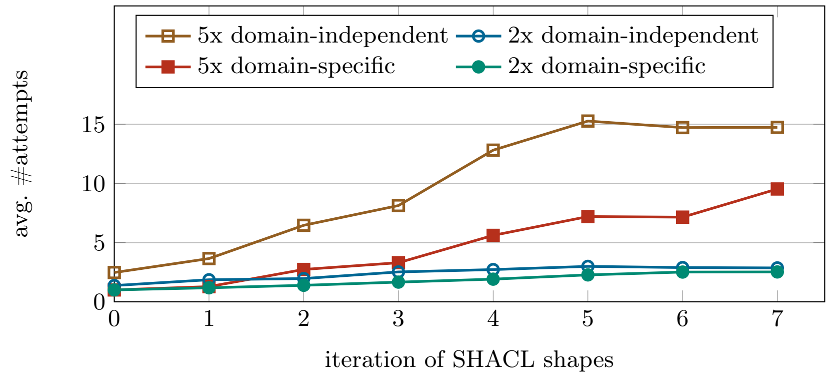


Fig. 5: Necessary attempts to generate a KG conforming to the provided shapes.

The results of the experiments are shown in Figure 4. The plot on the top shows the relation between the generation time and the number of applied mutations. Nearly all generations are performed within the time limit of 10s and the vast majority of mutants is generated in under 1s. Note that this is also true, if the number of mutation operators that are applied exceeds the number used in the case study (30) by a factor of three. As expected, there is a correlation between the number of applied mutation operators and the generation time. Interestingly, there is a huge spread in computation times, note that the y-axis is logarithmic. For a given number of applied mutation operators, the generation times often differ by more than two orders of magnitude.

The spread in generation times can partially be explained with the data from the second experiment, which is shown in the lower plot of Figure 4. Note, that both axis are logarithmic. Remember, that we applied a constant number of 30 mutation operators in this experiment. We see a strong correlation between the size of the input KG and the generation time. Still, the computation times vary considerably by a factor of about 10 for a given size of input KG. This shows that the structure of the specific KG and the applied mutation operators must also have a significant impact on the generation time.

The time to generate a valid KG depends on the concrete inputs and parameters, for example, shapes and number of mutations. However, based on the above results, we consider the time to generate a valid KG acceptable (<1s), and in practice we were able to generate sufficient KGs in a short time.

Attempts to Generate Valid Mutants. RDFMutate is intended to be used to generate KGs that conform to provided SHACL shapes. The stronger the restrictions expressed by the shapes are, the harder it is to find a mutant KG that conforms to them. Hence, the more sequences of operators have to be tried until a valid KG is found. We use the testing campaign of the tool *Suave* [14] for this evaluation as it provides a sequence of increasingly more strict SHACL shapes. Furthermore, the ontologies contained in the generated KGs are required to be consistent for this application. For each shape, we investigate four scenarios by varying the used mutation operators (domain-independent vs. domain-specific) and the number of applied mutations (two vs. five). In the original testing campaign, a mixture of both types of mutation operators was used and two operators where applied to generate a mutant. We generated 100 valid mutant KGs for each data point and calculated the average number of attempts.

Figure 5 show the result of this experiment. The shape has a huge impact on the number of attempts that are necessary to generate a conforming mutant. As expected, a stricter shape requires more attempts. The more operators are applied, the more attempts are necessary as the mutant KG differs more from the seed KG, which conforms to all of the shapes. Interestingly, the types of the mutation operators also have a large impact. We see that operators specifically designed for the application domain require fewer attempts and thus speed up the generation of valid mutant KGs.

Overall, it is hard to make a definitive statement about how hard it is to generate a mutant that is valid as this depends on the specific shapes, input KG and mutation operators. However, the limited number of attempts needed seem feasible in practice.

5 Discussion

RDFMutate is the first proposal of a mutation-based KG generator and has some limitations that we are aware of. In particular, we expect users of RDFMutate to have very specific constraints on the KGs for their software application, so they are interested in implementing custom extensions. Our developer documentation on github provides instructions on how to implement possible extensions [19].

5.1 Limitations and Extensions

RDFMutate only provides a simple, random generation strategy to select the mutation operators that get applied. Other tools that generate random KGs have more sophisticated search algorithms to find valid KGs [4, 35]. However, deciding if there is a sequence of mutations such that the mutant KG is valid is undecidable [10]. This makes finding an optimal, general strategy difficult, so users may extend RDFMutate by custom strategies for specific domains.

We distribute RDFMutate with four different reasoners to check the consistency of the ontology in the generated KG. Users might extend the selection by custom reasoners, e.g., by the reasoner that is later used in their application.

RDFMutate only supports one input format (plus implementation in Kotlin) to specify custom mutation operators, which is based on SWRL rules. Alternative formats might be of interest, e.g. SPARQL queries using an `update` term. To add support, one needs to extend both frontend and backend.

5.2 Sustainability and Future Work

RDFMutate is part of the SMOL [21] ecosystem (`smolang.org`), which aims to develop techniques and tools to increase the reliability of semantic web applications. To make usage of RDFMutate as easy as possible, we provide instructions for different methods to run RDFMutate: a JAR-file, a docker image and built instructions for the source code. Additionally, we not only provide a user documentation, but also a developer documentation that explains how others can extend and customize RDFMutate. We aim to use feedback from the community to continuously improve RDFMutate. In particular, we welcome contributions on GitHub in the form of issues and pull request to overcome limitation of RDFMutate. We are looking forward to incorporating extensions that proved to be useful for KGs in some domains to make them available for all users.

Further Use Cases. RDFMutate does not allow to specify parameters of the generated KGs, e.g. the number of nodes or the density of the relations, which limits its usage as a general synthetic data generator. However, we see use cases, apart from testing applications, for which the mutation-based generation is applicable.

In testing itself, further use cases remain to be explored as well, for example, using RDFMutate to perform metamorphic testing [3] or for quality control of ontologies. As discussed in the related work (see Section 2), mutations of axioms in OWL ontologies can be used as a technique for quality control [1, 31]. The idea is to investigate if the answers to predefined queries change when the ontology is mutated, which is expected for well-crafted ontologies. We are not aware of usage of this idea in recent years, which may be due to a lack of tool support. RDFMutate fills this gap and might help ontology engineers to apply the proposed approaches by being easily able to create mutant ontologies.

6 Conclusion

RDFMutate is the first tool for mutation-based KG generation. It allows the user to generate KGs for constrained input domains by defining custom mutation operators and constraints. Test data generated by RDFMutate was used to find bugs in widely-used applications and demonstrated that the generation times are short enough to use RDFMutate in practice. In the future, we aim to investigate test case generation for more KG-specific tasks, such as entity alignment, fusion, and generating synthetic data for training embeddings.

Resource Availability Statement. Source code for RDFMutate, data and source code to reproduce the evaluation experiments and the documentation are available from github [18, 19]. Additionally, we made RDFMutate permanently available on Zenodo [17]. Some ontologies for our evaluation are from the ORE 2014 reasoner competition dataset, which is available from Zenodo [27].

Acknowledgments This work was partially supported by the EU projects REMARO (956200) and SM4RTENANCE (101123490).

References

1. Bartolini, C.: Mutating owls: Semantic mutation testing for ontologies. In: Calabrò, A., Lonetti, F., Marchetti, E. (eds.) Proceedings of the International Workshop on domAin specific Model-based AppRoaches to vErificaTion and validaTiOn, AMARETTO@MODELSWARD 2016, Rome, Italy, February 19-21, 2016. pp. 43–53. SciTePress (2016). https://doi.org/10.5220/0005844600430053
2. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Symposium on Operating Systems Design and Implementation (OSDI 2008). pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
3. Chen, T.Y., Cheung, S., Yiu, S.: Metamorphic testing: A new approach for generating next test cases. CoRR **abs/2002.12543** (2020), https://arxiv.org/abs/2002.12543
4. Feng, Z., Mayer, W., He, K., Kwashie, S., Stumptner, M., Grossmann, G., Peng, R., Huang, W.: A schema-driven synthetic knowledge graph generation approach with extended graph differential dependencies ($\mathrm{GDD}^{\mathrm{X}}$s). IEEE Access **9**, 5609–5639 (2021). https://doi.org/10.1109/ACCESS.2020.3048186
5. Fioraldi, A., Maier, D.C., Eißfeldt, H., Heuse, M.: AFL++: Combining Incremental Steps of Fuzzing Research. In: Workshop on Offensive Technologies (WOOT). USENIX Association (2020), https://www.usenix.org/conference/woot20/presentation/fioraldi
6. Foundation, A.S.: Apache Jena, available at: https://jena.apache.org/, accessed 01-Mai-2025
7. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: Hermit: An OWL 2 reasoner. Journal of Automated Reasoning **53**(3), 245–269 (2014). https://doi.org/10.1007/S10817-014-9305-1
8. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Conference on Programming Language Design and Implementation (PLDI). pp. 213–223. ACM (2005). https://doi.org/10.1145/1065010.1065036
9. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: Whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012). https://doi.org/10.1145/2093548.2093564
10. Hariri, B.B., Calvanese, D., Montali, M., Giacomo, G.D., Masellis, R.D., Felli, P.: Description logic knowledge and action bases. J. Artif. Intell. Res. **46**, 651–686 (2013). https://doi.org/10.1613/JAIR.3826
11. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A semantic web rule language combining OWL and RuleML. W3C member submission, W3C (May 2004), https://www.w3.org/submissions/SWRL/
12. Horrocks, I., Wu, Z., Grau, B.C., Fokoue, A., Motik, B.: OWL 2 web ontology language profiles (second edition). W3C recommendation, W3C (Dec 2012), https://www.w3.org/TR/2012/REC-owl2-profiles-20121211
13. Hubert, N., Monnin, P., d'Aquin, M., Monticolo, D., Brun, A.: PyGraft: Configurable generation of synthetic schemas and knowledge graphs at your fingertips. In: Proc. 21st International Conference on The Semantic Web (ESWC 2024). Lecture Notes in Computer Science, vol. 14665, pp. 3–20. Springer (2024). https://doi.org/10.1007/978-3-031-60635-9_1
14. John, T., Johnsen, E.B., Kamburjan, E.: Mutation-based integration testing of knowledge graph applications. In: Proc. 35th IEEE International Symposium on

Software Reliability Engineering (ISSRE 2024). pp. 475–486. IEEE (2024). https://doi.org/10.1109/ISSRE62328.2024.00052

15. John, T., Johnsen, E.B., Kamburjan, E.: ESE 2025 knowledge graph mutation virtual machine (Feb 2025). https://doi.org/10.5281/zenodo.14899988
16. John, T., Johnsen, E.B., Kamburjan, E.: Mutation-based testing of knowledge graph applications (2025), submitted
17. John, T., Johnsen, E.B., Kamburjan, E.: RDFMutate (ISWC 2025 submission) (May 2025). https://doi.org/10.5281/zenodo.15394157
18. John, T., Johnsen, E.B., Kamburjan, E.: RDFMutate repository. https://github.com/smolang/RDFMutate (2025)
19. John, T., Johnsen, E.B., Kamburjan, E.: RDFMutate wiki. https://github.com/smolang/RDFMutate/wiki (2025)
20. John, T., Johnsen, E.B., Kamburjan, E., Steinhöfel, D.: Language-based testing for knowledge graphs. In: Proc. 22st International Conference on The Semantic Web (ESWC 2025). Lecture Notes in Computer Science, vol. 15719, pp. 24–46. Springer (2025). https://doi.org/10.1007/978-3-031-94578-6_2
21. Kamburjan, E., Klungre, V.N., Schlatte, R., Johnsen, E.B., Giese, M.: Programming and debugging with semantically lifted states. In: The Semantic Web - 18th International Conference, ESWC 2021. Lecture Notes in Computer Science, vol. 12731, pp. 126–142. Springer (2021). https://doi.org/10.1007/978-3-030-77385-4_8
22. Kazakov, Y., Krötzsch, M., Simancik, F.: The incredible ELK - from polynomial procedures to efficient reasoning with EL ontologies. Journal of Automated Reasoning **53**(1), 1–61 (2014). https://doi.org/10.1007/S10817-013-9296-3
23. Keet, C.M., Lawrynowicz, A.: Test-driven development of ontologies. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9678, pp. 642–657. Springer (2016). https://doi.org/10.1007/978-3-319-34129-3_39, https://doi.org/10.1007/978-3-319-34129-3_39
24. Lanthaler, M., Cyganiak, R., Wood, D.: RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C (Feb 2014), https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/
25. Lee, S., Bai, X., Chen, Y.: Automatic mutation testing and simulation on OWL-S specified web services. In: Proceedings 41st Annual Simulation Symposium (ANSS-41 2008), April 14-16, 2008, Ottawa, Canada. pp. 149–156. IEEE Computer Society (2008). https://doi.org/10.1109/ANSS-41.2008.13
26. Louadah, H., Papadakis, E., Mccluskey, T.L., Tucker, G., Hughes, P., Bevan, A.: Translating ontological knowledge to pddl to do planning in train depot management operations. In: 36th Workshop of the UK Planning and Scheduling Special Interest Group. AAAI press (2021)
27. Matentzoglu, N., Parsia, B.: ORE 2014 reasoner competition dataset (Jul 2014). https://doi.org/10.5281/zenodo.10791
28. Miller, B.P., Fredriksen, L., So, B.: An Empirical Study of the Reliability of UNIX Utilities. Commun. ACM **33**(12), 32–44 (1990). https://doi.org/10.1145/96267.96279
29. Openllet, issue tracker, https://github.com/Galigator/openllet/issues, last accessed on 30.04.2025

30. Parsia, B., Matentzoglu, N., Gonçalves, R.S., Glimm, B., Steigmiller, A.: The OWL reasoner evaluation (ORE) 2015 competition report. Journal of Automated Reasoning **59**(4), 455–482 (2017). `https://doi.org/10.1007/S10817-017-9406-8`
31. Porn, A.M., Peres, L.M.: Semantic mutation test to OWL ontologies. In: Hammoudi, S., Smialek, M., Camp, O., Filipe, J. (eds.) ICEIS 2017 - Proceedings of the 19th International Conference on Enterprise Information Systems, Volume 2, Porto, Portugal, April 26-29, 2017. pp. 434–441. SciTePress (2017). `https://doi.org/10.5220/0006335204340441`
32. Portisch, J., Paulheim, H.: The DLCC node classification benchmark for analyzing knowledge graph embeddings. In: Sattler, U., Hogan, A., Keet, C.M., Presutti, V., Almeida, J.P.A., Takeda, H., Monnin, P., Pirrò, G., d'Amato, C. (eds.) Proc. 21st International Semantic Web Conference (ISWC 2022). Lecture Notes in Computer Science, vol. 13489, pp. 592–609. Springer (2022). `https://doi.org/10.1007/978-3-031-19433-7_34`
33. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2005). pp. 263–272. ACM (2005). `https://doi.org/10.1145/1081706.1081750`
34. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Journal of Web Semantics **5**(2), 51–53007 (2007). `https://doi.org/10.1016/J.WEBSEM.2007.03.004`
35. Vecovska, M., Jovanovik, M.: Rdfgraphgen: A synthetic RDF graph generator based on SHACL constraints. CoRR **abs/2407.17941** (2024). `https://doi.org/10.48550/ARXIV.2407.17941`
36. Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: The Fuzzing Book. CISPA Helmholtz Center for Information Security (2024), `https://www.fuzzingbook.org/`, retrieved 2024-07-01 16:50:18+02:00