Testing Knowledge Graph Applications

Tobias John
Einar Broch Johnsen
Eduard Kamburjan
Dominic Steinhöfel

ISWC, Nara, 02.11.2025

IT UNIVERSITY OF COPENHAGEN

Welcome!

What is this tutorial about?

- A *Software Testing* perspective on the semantic web
- Focus: Knowledge graph generation for *automated* software testing
- Aim: You will be able to design and develop better test suites for your tools

Welcome!

What is this tutorial about?

- A *Software Testing* perspective on the semantic web
- Focus: Knowledge graph generation for *automated* software testing
- Aim: You will be able to design and develop better test suites for your tools

Content

- Different test oracles and generation approaches specific to KGs
- Improve a test suite for `BuggyTable`
- `BuggyTable` takes a FOAF graph and a name, and generates CSV for all persons what share an organization with the named person
- At least 3 bugs are injected (there will be more)
- Grand Prize for participant who finds most bugs

Agenda

9AM Session 1

- Testing Theory
- Setup
- Mutation-based fuzzing

11AM Session 2

- Setup
- Grammar-based fuzzing
- Outlook on further tools

Prereqs.

Please install

- Java/Kotlin IDE and compiler
- Git
- Python

Software Turtles all the way down

Application Quality

Application quality is determined by data quality, software quality and their integration.

Software Turtles all the way down

Application Quality

Application quality is determined by data quality, software quality and their integration.

Software Quality is Important

- Five papers in high-impact venues (including nature) retracted after a bug in python implementation of analysis algorithm

  [Miller, *Software problem leads to five retractions.*, 2007]

- Faulty analysis leads to wrong data basis for decision about austerity in Europe

  [Herndon et al., *Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff*, 2013]

- "Replication crisis" w.r.t. Jupyter notebooks: less than 25% are runnable

  [Pimentel et al., *Understanding and improving the quality and reproducibility of Jupyter notebooks*, 2021]

- Automated testing found bugs in HermiT, ELK, Openllet, Apache Jena and OWL-API frontends, and robotic simulators [John et al. *ESWC'25,ISSRE'24*]

Agenda

1. Background on automated testing from a KG perspective
2. Some recent results
3. Mutation-based testing with tasks
4. Language-based testing with tasks
5. Conclusion

Testing

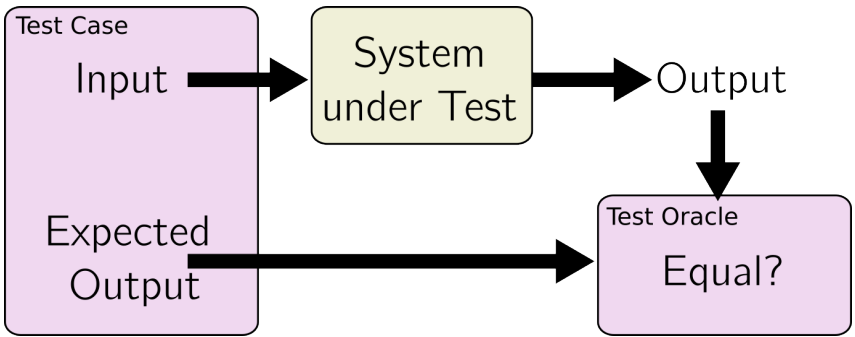
Reminder: Software Testing

Testing Level

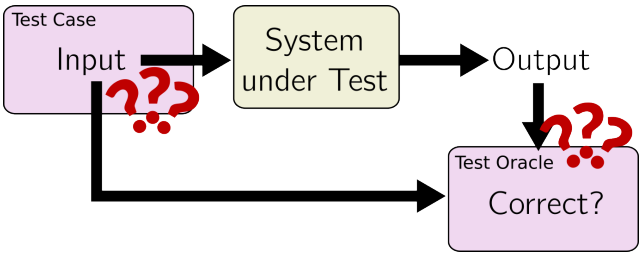
- Unit testing: test a single software component: function, class,...
- Integration testing: test the interface between two or more software components
- End-to-end (E2E) testing: test whole application

- KG can be considered either an own component or *input*
- We will develop E2E tests and consider the KG an input

Reminder: Software Testing



Automated testing



- Manual testing (a) is time-intensive and (b) relies on developers being able to identify corner cases
- How to automatically generate test cases?
- Challenge 1: How to generate inputs?
- Challenge 2: How to validate previously unknow input?

- Clone `BuggyTable`
- `git clone git@github.com:Edkamb/BuggyTable.git`
- (Demo BuggyTable)

Task 0a: Run predefined mutations (5min)

- Go to the `BuggyTable` project
- Inspect `mutate/config1.yml`
- Inspect `ontologies/onto1.ttl`
- Run `java -cp .:mutate/rdfmutate-1.1.1.jar org.smolang.robust.MainKt --config=mutate/config1.yaml`

Test Oracles

Consider a program $f : I \to O$ (Examples in the `BuggyTable`, demo)

- Differential testing: There is a second program g with $\forall x.\ f(x) \sim g(x)$
  In the case where functionality is reimplemented: $X \subseteq I, \forall x \in X.\ f(x) = g(x)$
- Property-based testing
  - Input-independent: There is a property ϕ with $\forall x. f(x) \models \phi$
  - Input-dependent: There is a property $\phi(i)$ with $\forall x.\ f(x) \models \phi(x)$
    Property $\phi(i)$ should be a lot easier to compute than $f(i)$
- Metamorphic testing: There is a property that relates several runs of f.
  For example $\forall x. f(x) = f(f^{-1}(f(x)))$

Test Case Generation

If the input format is complex and structured (such as KG), random input will most likely fail to detect bugs beyond input validation

Mutation-based fuzzing

- If example inputs are available, define operations to change them slightly
- Challenge: Operations may be specific for application to make sense

(Beware: Mutation-based fuzzing $\neq$ mutation-based testing)

Schema-based fuzzing

- If an input specification is available, define a generator for random instances
- Special case where input is defined by grammar: language-based fuzzing

(Demo BuggyTable TestSuite)

Success Stories in Automated Testing of KGs: Schema-Based

Automated Testing

- Language-based approach to generate random graphs and ontologies with ISLa
- Two grammars: RDF/TTL and OWL functional syntax

```
<ontology>       ::= "Ontology (" <declarations> " " <axioms> ")"
<axioms>         ::= <axiom> | <axiom> "\n" <axioms>
<axiom>          ::= <classAxiom> | <assertion> | <dataTypeDefinition> | [...]
[...]
<literal>        ::= <typedLiteral> | <stringNoLang> | <stringWithLang>
<stringWithLang> ::= <QuotedString> <LanguageTag>
```

Targets

- RDF/TTL parser and frontend utilities of Apache Jena and OWL-API
- Three OWL-EL reasoners via differential testing

[John, Kamburjan et al. *Language-Based Testing of Knowledge Graphs*, ESWC'25]

Automated Testing: Frontend Bugs

- First bug found in RDF 1.2 TTL standard with second generated file

```
<P:A> <B> <C>.
@prefix P: <http://test.no#>
```

- Both parser have bugs in corner cases, despite a formal grammar in the standard!

```
<A> <B> -.7 . // fails to parse literal
<A> <B> ; ; . // fails to parse double empty list
```

- OWL-API profile checker rejects all OWL-EL ontologies that use language tags

```
<A> <B> "test"^^xsd:String@dk_DK
```

Automated Testing: OWL-EL Reasoners

Test Targets

Three reasoners included by default with Protege

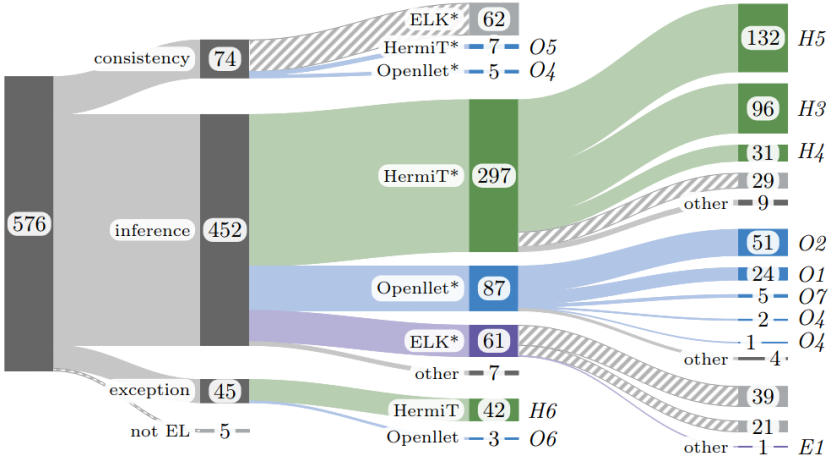
- HermiT (v.1.4.5.519)
- Pellet/Openllet (v.2.6.5)
- ELK (v.0.6.0)

Test Procedure

- Generate new ontology, and ask all three reasoners if it is consistent and to derive all possible axioms
- If results are different (or exception is thrown), investigate
- Extra tool to reduce ontology by axiom pinpointing

Automated Testing: OWL-EL Reasoners

- Found and reported 15 bugs, 13 from failed logical inference, 2 from exceptions
- Language tags and corner cases in the hierarchy



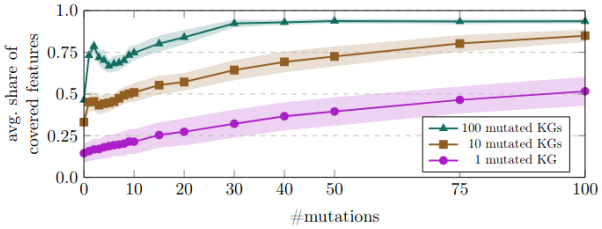
Automated Testing: OWL-EL Reasoners

```
1 //ELK classifies as inconsistent
2 Prefix(:=<http://www.example.org/reasonerTester/>)
3 Ontology (
4     Declaration(Class(:B))   Declaration(Class(:A))
5     Declaration(DataProperty(:dr))   Declaration(NamedIndividual(:a))
6     EquivalentClasses( DataHasValue(:dr "s1"@fr)   :A   :B )
7     DisjointClasses(   DataHasValue(:dr "s1"@en)   :A )
8     ClassAssertion(:B :a))
```

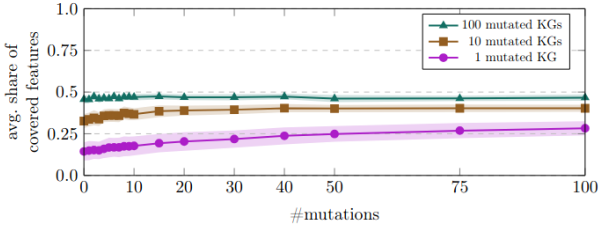
```
1 //HermiT fails to derive DataPropertyAssertion(:dp :a "data")
2 Prefix(:=<http://www.example.org/reasonerTester/>)
3 Ontology (
4     Declaration(DataProperty(:dp))  Declaration(NamedIndividual(:a))
5     EquivalentClasses( ObjectOneOf(:a)  DataHasValue(:dp "data") ))
```

Success Stories in Automated Testing of KGs: Mutation-Based

Integration Testing: Input Coverage



(a) Domain-specific mutation operators



(b) Learned operators

- Input feature coverage: How many features are used?
- Measured via OWL vocabulary
- Domain-specific operators can be used to force feature interactions

Integration Testing: Results

Targets

- SUAVE: Simulator for self-adaptive AUV based on ROS
- GeoSimulator: Simulator for geological process based on geological ontologies
- OWL-EL reasoners: Same setup

Seed Ontologies

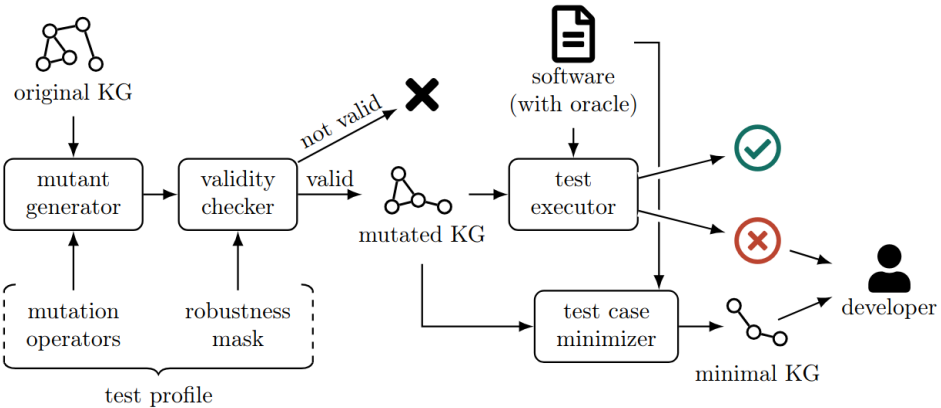
- Suave and GeoSimulator: Only one ontology as default example
- OWL-EL reasoners: 307 Ontologies from latest OWL reasoning competition

Results

- SUAVE: Mistakes in OWL modeling
- GeoSimulator: No bugs
- OWL-EL reasoners: 6 additional bugs related to reasoning over class hierarchies

Mutation-Based Fuzzing

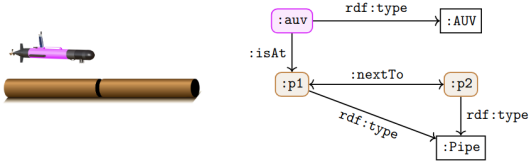
Structure


original KG
mutant generator
validity checker
not valid
valid
mutated KG
software (with oracle)
test executor
mutation operators
robustness mask
test profile
test case minimizer
minimal KG
developer

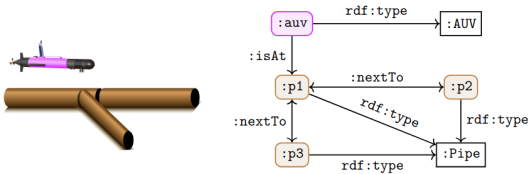
Generic and Domain-Specific Mutation Operators

Mutation Operators

- A mutation operator defines a minor change in a knowledge graph
- A generic mutation operator works on all graphs. For example, removing a arbitrary triple
- A domain-specific mutation operators works on graphs with a certain vocabulary. For example, removing a subgraph modeling a whole object



(c) Scenario with corresponding KG representation before mutating.



(d) Scenario with corresponding KG representation after mutating.

Mutation Operators: Kotlin and SWRL syntax

```
1 class PizzaMutation(model: Model) : Mutation(model) {
2     ...
3
4   override fun createMutation() {
5     val toppings = model.listResourcesWithProperty(RDF.type, topping)
6     val t = toppings.toSet().random()
7     val p = model.createResource(":newPizza" + Random.nextInt())
8     addSet.add(model.createStatement(p, RDF.type, pizza))
9     addSet.add(model.createStatement(p, hasTopping, t))
10    super.createMutation()
11  }
12 }
```

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

(Demo RDFMutate FOAF)

Extraction and Robustness

- Developing Domain-Specific Operators requires manual effort.
- We include an extraction tool based on association rule mining (via RDFRules).
- Per mined rule, we add two operators: addition and removal
- To run the operation extraction, run the following

```
java -jar mutate/rdfmutate-1.1.1.jar --operator-extraction
--config=mutate/extract/extract.yaml
```

- Robustness masks filter mutants and specify the interface to the software
- For example, mutation of TLOs rarely useful
- Specified using SHACL shapes

Usage

```
seed_graph:
...
output_graph:
...
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
```

Task 1a: Create your first mutation operator! (10min)

- Clone `RDFMutate` and checkout the `tutorial` branch
- `git clone git@github.com:smolang/RDFMutate.git`
- `git checkout tutorial`
- Create operators that add and remove an organization under `src/main/kotlin/org/smolang/robust/domainSpecific/foaf/Mutants.kt`
- Compile rdfmutate (`./gradlew build -x test`)
- In `BuggyTable`, modify `mutate/config2.yaml` that uses these mutations to generate 10 input files
- Run `java -cp .:</path/to/rdfmutate>/build/libs/rdfmutate-1.1.1.jar org.smolang.robust.MainKt --config=mutate/config2.yaml`

Note: do not modify the model parameter

Task 2a: Create better mutation operators! (20min)

- Clone `BuggyTable`
- `git clone git@github.com:Edkamb/BuggyTable.git`
- Create operators that add a homepage or mail address to a person
- Create operators that add or remove a person to an organization
- Create an operator that replaces a name with a random string literal
- Compile rdfmutate (`./gradlew build -x test`)
- In `BuggyTable`, modify `mutate/config2.yaml` that uses these mutations to generate 10 input files
- Run `java -cp .:</path/to/rdfmutate>/build/libs/rdfmutate-1.1.1.jar org.smolang.robust.MainKt --config=mutate/config2.yaml`

Note: do not modify the model parameter

Task 3a: Integrate (10min)

- Create a CSV file that lists all your created files, and the name of a person within
- Optional: Create a script that automatically generates such CSV files
- Add this CSV to the test suite
- Happy debugging :)

Grammar-Based Fuzzing

Input Specification Language (ISLa)

- A grammar-based fuzzer instantiates syntactic terms according to a grammar
- What about semantic constraints? Variables must be declared before used, etc.
  1. Encode semantic constraints into grammar
  2. Ignore constraints and use them as a filter afterwards
  3. Use a fuzzer that considers semantic constraints

Chossing your Fuzzer

The more complex your fuzzing specification becomes, the more time you spend figuring out whether the test fails because of your setup or your program

- There is per se nothing wrong with specializing your grammar
- Constraints may slow down your solver and introduce bias into input generation

Demo

- rdf.bnf/ISLa
- simple_rdf.bnf/isla

Task 0b: Run predefined grammar (5min)

- Install ISLa according to its instructions (you do not need to install csvlint etc.)
- `https://github.com/rindPHI/isla`
- `python -m islavenv islavenv`
- `source islavenv/bin/activate`
- `pip install --upgrade pip`
- `pip install isla-solver`
- Generate random RDF: `isla solve </path/to/buggyTable/mutate/rdf.bnf>`

Program Specification

- Program specifications give a second perspective on the program
- The more different the perspective the better
- An input grammar is a precondition, oracle is a postcondition
- The included grammar is a fragment from the RDF TTL 1.2 standard
- Contains the usual forgotten features: language tags etc.

ISLa Input Files

```
<start>        ::= <turtleDoc> <triple> ".\\n" <triple> ".\\n"
<turtleDoc>    ::= <statement> | <statement> <turtleDoc>
<statement>    ::= <directive> | <triple> ".\\n"
<directive>    ::= "@prefix " <PNAME_NS> <IRIREF> ".\\n"
<triple>       ::= <subject> " " <predicate> " " <object>
<subject>      ::= <IRI>      <predicate> ::= <IRI>     <object> ::= <IRI>
<IRI>          ::= <PNAME_LN>
<IRIREF>       ::= "<b>" | "<e>"   <PNAME_NS> ::= "p1:" | "p2:"
<PNAME_LN>     ::= <PNAME_NS> "c" | <PNAME_NS> "d"
```

```
( forall <PNAME_NS> x in <triple>:
  exists <directive> y = "@prefix {<PNAME_NS> z}<IRIREF>.\n": x=z )
and
( forall <directive> a = "@prefix {<PNAME_NS> xy}<IRIREF>.\n":
    forall <directive> b = "@prefix {<PNAME_NS> yy}<IRIREF>.\n":
      (xy = yy implies same_position(a,b)) )
```

Task 1b: Spezialize your Grammar (Vocabulary-Based, 10min)

Approach 1: Vocabulary

- Allow any RDF, but restrict the vocabulary
- May generate insights for SHACL shapes – what exactly is our input specification?

- Modify `simple_rdf.bnf` so that it only generates triples using our vocabulary
- To do so, introduce use lists instead of `<ITI>` for each position in the triple
- Generate a new input and load it using
  `python load.py <output.ttl> full.sparql`

Task 2b: Spezialize your Grammar (Structure-Based, 20min)

Approach 2: Structure

- Allow only specific RDF, but reuse subclauses from RDF grammar
- Needs more work, but gives you more control and is more precise

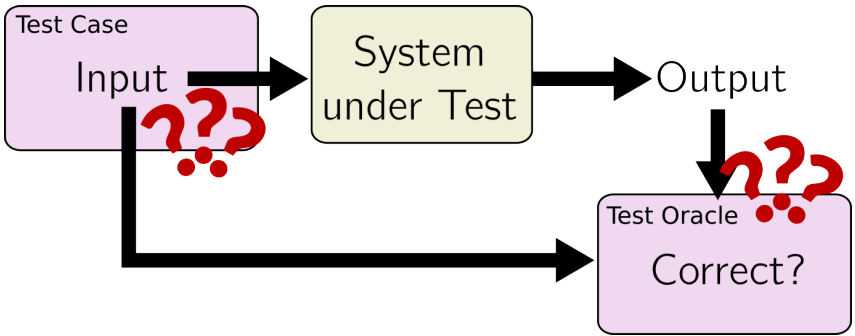
- Investigate `simple_foaf.bnf`. It only generates new organizations.
- Modify `simple_foaf.bnf` so that it also generates persons. Reuse the literal non-terminals.
- Create a constraint to ensure that if a person is member of an organization, then the organization has a name declared.
- `simple_bnf.isla` can be a starting point.
- Generate a new input and load it using
  `python load.py <output.ttl> full.sparql`

Task 3b: Integrate (10min)

- Create a CSV file that lists all your created files, and the name of a person within
- Optional: create a tool that creates n new inputs and automatically generates such CSV files
- Add this CSV to the test suite
- Happy debugging :)

Conclusion

Automated Testing



More Testing

Integration

- Do not generate random tests every time you build the tool
- Instead: extra tasks that occasionally run large testing campaigns
- Afterwards: detect root causes, use tests for confirmed bugs as regressions

Other Tools

- RDFMutate and ISLa were the tools we used in our studies
- There are numerous other generation tools for schema-based generation
- Including some for RDF, based on SHACL or graph transformations
- These may optimize towards regular, pre-defined structures, not corner-cases

Outlook

Where is the coverage?

- Both approaches are black-box, and do not use knowledge about the code
- Code coverage found to be bad indicator in compilers, reasoners, databases...
- Especially if logic is in queries, not statements

Outlook

Where is the coverage?

- Both approaches are black-box, and do not use knowledge about the code
- Code coverage found to be bad indicator in compilers, reasoners, databases...
- Especially if logic is in queries, not statements

- **What are best practices for software testing of knowledge graph applications?**
- How to pick an RDF generator? No comparative studies available
- What notions of input coverage are useful?

Outlook

Where is the coverage?

- Both approaches are black-box, and do not use knowledge about the code
- Code coverage found to be bad indicator in compilers, reasoners, databases...
- Especially if logic is in queries, not statements

- **What are best practices for software testing of knowledge graph applications?**
- How to pick an RDF generator? No comparative studies available
- What notions of input coverage are useful?

Thank you for your attention!