Language-Based Testing for Knowledge Graphs

Tobias John[1], Einar Broch Johnsen[1],
Eduard Kamburjan[2,1], and Dominic Steinhöfel[3]

[1] University of Oslo, Oslo, Norway
{tobiajoh,einarj}@ifi.uio.no
[2] IT University of Copenhagen, Copenhagen, Denmark
eduard.kamburjan@itu.dk
[3] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
steinhoefel@cispa.de

Abstract. Knowledge graphs rely on a vast ecosystem of software tools, such as parsers, APIs and reasoners. Yet, tool developers have little support to ensure tool reliability. Here, we demonstrate how recent advances in test case generation for highly structured and constrained inputs can support software developers in the semantic web field. We develop input generators for RDF Turtle and the OWL EL profile, and report on numerous bugs we found in parsers, reasoners and APIs of widely used tools and libraries, as well as imprecisions in standards and documentation. We provide actionable insights on using automated testing to increase the reliability of software tools for knowledge graphs.

Keywords: Automated Testing · RDF · OWL EL · Reasoning.

1 Introduction

Knowledge graphs (KGs) [45] are used in business-critical applications, supported by a rich ecosystem of standards, technologies and software tools [39]. Nevertheless, quality considerations are mostly concerned with data or ontology quality [103, 107], and rarely consider the software tools that underlie the application. Software reliability, however, is as important as the quality and reliability of the data that this software processes to establish reliability and maintainability of the overall application. Indeed, the increasing use of digital methods in other fields heavily relies on correct and reliable software. Misuse of digital tools and software can lead to unexpected threats to validity and has already had consequences in fields such as, e.g., biology, economics, physics, end anthropology [24, 67, 92]. KGs, with their wide range of applications, should provide confidence when used—especially if they themselves are used to improve confidence in an overall application, e.g., to counteract gaps or hallucinations in neuro-symbolic AI [26, 97].

In the semantic web field, the challenges of integrating and maintaining KG tools, such as reasoners, are well recognized [43, 52, 74], yet there is little support for the tool developers. To tackle this problem, we here investigate the applicability of the most widespread and important software quality method: *Testing.*

We propose a method to automate the testing procedure by generating input data [3] for software that operates on KGs and ontologies. Automated test case generation has found bugs in very mature, yet highly complex software classes: SMT solvers [13, 106], databases [2, 5], compilers [18], and others [64].

KGs pose a challenge for input generation, even if restricted to RDF and associated technology, such as OWL: They are highly structured, yet subject to numerous constraints. For example, if the generated test cases are to target a certain OWL profile, one needs to impose restrictions on the ontology used as a test case. We base our work on recent advances in language-based approaches to automated testing, namely grammar-based fuzzing with complex semantic invariants [93, 94]. *Fuzzing* techniques massively generate test inputs to cover as much as possible of the input space. With the notable exception of symbolic execution-based white box techniques, this is done with little to no knowledge about the system-under-test [112].

In this paper, we present two input grammars with constraints that generate either arbitrary RDF or an ontology in the OWL EL profile, and generate random inputs based on these constraint grammars for two different studies.

RDF-TTL: The first grammar generates arbitrary RDF graphs in Turtle syntax, which we use to find bugs in the frontend (parser and selected methods) of the often used OWL-API [47] and Apache Jena [31]. Here, we use as test oracle the generic oracle whether the application crashes with an exception.

OWL EL: The second grammar generates ontologies in the OWL EL profile, which we use to find bugs in the EL reasoners bundled with the Protege editor [68]: HermiT [35], Pellet [89], and ELK [59]. Here, we use differential testing [90]: The ontology is analyzed by all reasoner tools, which must return the same result. If the reasoners do not agree, we investigate manually whether the disagreement is due to a completeness or soundness issue.

Even in the absence of precise testing oracles, we found bugs in all considered tools, as well as imprecisions in error reports and the RDF-Turtle standard; these bugs have been reported and partially fixed by the respective maintainers. Our results demonstrate that language-based testing can already be used to increase and investigate the quality and reliability of software for KGs.

We hope that this study will contribute to increase the software quality of both existing and newly developed tools. To this end, we discuss both our findings in light on the specifics of the field, and give actionable insights for further studies. To summarize, our main contributions are (1) the first systematic study on finding bugs in KG applications based on language-based testing, (2) a documentation of several found bugs in mature applications operating on RDF and OWL, and (3) two input grammars for further studies and (4) actionable insights for improving the reliability of KG applications.

Paper overview. This paper is structured as follows. Section 2 discusses related work, background on testing, and introduces the used ISLa tool [93], Section 3 gives two examples for input grammars and test oracles. Section 4 reports on the found bugs, Section 5 discusses our findings and Section 6 concludes.

2 Background

2.1 Fuzzing: Breaking Things With Random Inputs

Testing is concerned with running a program with a *test input* and examining the program's behavior. If the program does not *behave as expected*, you have discovered a bug. The hidden challenges in this simple principle are: (1) *How do we find test inputs that effectively trigger bugs?* and (2) *How do we tell whether a behavior is expected?* The first of these problems is the *test generation problem.* The second is the *oracle problem.* Let us take a look at test input generation. What is so difficult about that? At first sight, not much, really. If you do not want to generate test inputs by hand, you can use fully random inputs. This was exactly the idea behind the very first *fuzzer* by Miller et al. [66].

Miller and his team managed to crash roughly *every third* tested application, including tools like Emacs and the c-shell. This tremendous success does not mark the end of research on input generation. The problems Miller et al. discovered were relatively shallow—adding serious input validation would fix them. To find more complex bugs, we need test data that *reaches deeply into the structures of a program.* For example, to crash the program `if x == 17: fail()`, one needs to hit 1 out of 2^{64} Integers on a 64-bit system. Since Miller's days, fuzzing evolved. The most "heavyweight" successors are symbolic execution-based white box fuzzers [15, 36, 37, 86]. These fuzzers collect conditions along the path triggered by an input. By negating one of these conditions and using constraint solvers, they come up with an input following a different path. These fuzzers are incredibly powerful—which makes them incredibly difficult to build in a way that scales. The most popular approach these days follows a pragmatic middle way. Coverage-based evolutionary fuzzing [11,30] obtains "gray box" information from the tested program: Not detailed conditions, but some kind of coverage information. For example, code lines. These fuzzers mutate inputs they have seen before and retain those that cover new code.

Both black and gray box approaches suffer from a problem that their white box competition gracefully avoids: That of *fuzz blockers* [32]. To *not* crash the program `if x != 17: fail() ...` by reaching the first `fail()`, we again must hit 1 out of 2^{64} values for `x`. To meaningfully *continue* fuzzing, the fuzz blocker `fail()` must be fixed first. Otherwise, our fuzzing campaign cannot progress.

The second challenge in automated software testing is that we must distinguish successful from failing program runs—automatically. We call a procedure that performs this task a *test oracle.* Fuzzing usually relies on a very simple type of oracle. That a program crashes or seemingly fails to terminate while your computer's fans crank up is "obviously" bad.

Barr et al. [7] distinguish three categories of oracles. This "crash oracle" common in fuzzing is an example of an *implicit* test oracle that relies on general, implicit knowledge. Everyone who has written a unit test preceded by *assertions*, has used *specified* test oracles. The third category consists of *derived* test oracles; for example, *differential testing* [90] uses a different program that should behave the same as the program-under-test. While this requires to have two programs with the same functionality, it is commonly used by companies operating in safety-critical domains: They implement twin systems following the informal specification of the main system. However, those twins are implemented by different teams and in different technologies. If the original and the twin system behave differently, this is a failure. Finally, *metamorphic oracles* [19]—another kind of derived oracle—exploit properties of the tested program's domain. Commutativity of addition is such a property in the domain of a calculator. If a different results for $a + b$ and $b + a$ are returned, that is a bug.

2.2 Language-Based Testing with ISLa

One major challenge for fuzzers when it comes to complex tools such as compilers or solvers, is to generate *structured* input—feeding randomly generating strings to a parser will not result in a deep penetration of the program. The same holds for programs operating on knowledge graphs and ontologies, such as reasoners.

For this reason, there are numerous fine-grained variations of the general theme of black, gray, and white box fuzzers. Hybrid fuzzers [95, 110], for example, combine white and gray box approaches to perform well while still reaching well-guarded program parts. In this paper, we use a particular breed of black box fuzzer: a *specification-based* input generator. Like Miller et al., we treat the program as a black box. However, we exploit knowledge about the *input language* of a program to generate meaningful input. Specification-based testing optimizes *input space coverage*, assuming that this measure is correlated to code coverage [41]. The advantage of this method over white box methods is that it does not require complex instrumentation and constraint solving. On the other hand, evolutionary fuzzers frequently fail to reach program paths guarded by complex constraints (which motivated hybrid fuzzing). If we solve these constraints at the specification level, we can bypass this difficulty—and still find complex bugs.

A popular approach to specify input languages is to use *context-free grammars (CFGs).* [44, 46, 78] While CFGs have a formal methods background, they don't scare away mainstream programmers. Parser generators such as ANTLR and language descriptions in RFCs are all based on CFGs. A CFG-based fuzzer iteratively expands the start nonterminal symbol `<start>` of a CFG by picking a random expansion rule. This process stops when only terminal symbols that cannot be further expanded are left. For an arithmetic expression grammar, `<start>` might first become `<expr> + <expr>`, then `(<expr> * <expr>) + <num>`, etc., until finally resulting in a terminal expression like `(2 * (3 - 1)) + 4`.

Grammars are a beautifully simple formalism for specification-based fuzzing. However, beauty and simplicity never come for free. Imagine you want to test a C compiler with a CFG for C programs. Most of your C programs will result in an error message like "use of undeclared identifier." They contain expressions like `int x = y;` with a variable `y` that was not properly declared. You can still test the compiler's parser component, but not the more interesting, deeper layers.

ISLa (Input Specification Language) [93, 94] is a specification language and constraint solver that allows you to refine the language of your CFG with additional constraints. In the case of C programs, you might want to add the constraint "all identifiers used in an expression occur on the left-hand side of some previous declaration." The ISLa language supports "all" and "some" constraints over grammar symbols. In ISLa, they are called "forall" and "exists." An ISLa constraint for a simple C-like grammar could look as follows:

```
forall <var> vUse in <expr>: exists <decl> assgn:
    ( before(decl, vUse) and assgn.<lhs> = vUse )
```

An ISLa language specification consists of two parts: A *syntactic* one—the grammar—and a *semantic* one—the constraints. The ISLa language-based testing tool takes such a specification and produces inputs of the right structure that satisfy all the given constraints. In the C example, all variables inside `<expr>` trees are guaranteed to be declared in some `<decl>` tree occurring `before`. In this paper, we use ISLa grammars and constraints to specify the RDF Turtle syntax and the OWL functional syntax.

2.3 Related Work

Testing Graph Databases and Knowledge Graphs. In recent years, several methods have been proposed to test graph databases, mostly the underlying database management systems for labeled property graphs [50, 54, 58, 63, 113, 114, 116] or database management systems for RDF triple stores [108]. We are only aware of one method that tests applications working with RDF data [55].

All methods generate random graphs as input and most methods do this by generating random nodes and relations that are later randomly assigned labels. A few methods generate the random graph by applying a number of random mutations to an initial graph [55,63,116]. Most methods use all randomly generated knowledge graphs for testing. For systems with long run times, a mechanism to filter for and only use relevant KGs was proposed in [55].

Concerning the test oracle, most methods aim to identify *logic bugs*, i.e. bugs that do not result in any error message. To do so, differential testing [50,113,114] as well as metamorphic testing [54, 58, 63, 108, 116] are used. Competitions are also applying differential testing implicitly; in particular, the OWL Reasoner Evaluation in 2015 [75] reported several bugs in OWL reasoners discovered by different answers to the same input.

There is no approach to test application and tools for knowledge graphs that do not rely on the specific structure given by the database management systems.

Knowledge Graph Synthesis. Traditional methods for generating random graphs are usually based on the Erdős-Rényi model [28] or on the Barabási-Albert model [1]. Such methods are still used today and are able to generate very large graphs [17, 73]. However, they lack a representation for the semantics of the nodes and edges.

There are mainly two approaches to generate random knowledge graphs, i.e. graphs with semantic labeling: data-driven and schema-driven approaches. *Data-driven* approaches use existing data, such as tabular database data [38], execution traces [4] or existing reference graphs [20, 80, 84, 104, 109] to generate new graphs. Some of these approaches use generative neural networks for the generation [84, 104, 109]. *Schema-driven* approaches use provided schema information to generate graph data that conforms to the specified schema [6, 29, 102].

While most approaches are domain-independent, some are tailored towards specific applications, such as social networks [4, 76, 79] or molecular graphs [84]. We present in our work domain-independent generators.

We are only aware of two existing approaches that generate graphs that contain not only data but also schema information [51, 77]. Both approaches limit the type of schema information that can be encoded. In particular, some features of RDF and OWL are missing, such as data relations and language tags. Additionally, their set of operators does not represent any defined OWL profile.

To the best of our knowledge, all previous approaches for generating knowledge graphs build the graph structure explicitly; in contrast, our work relies on the provided grammar for the input format of the knowledge graph.

Testing. There are several areas of automatic test case generation that target software that share similarities with software for knowledge graphs.

As knowledge graphs are a way to structure data, testing of *database management systems* is closely related. While the test cases, i.e. databases, can be created based on schema information [49] or on provided queries [9, 23], finding an oracle to evaluate the test runs is a challenge. One of the earliest techniques is differential testing [90], which remains relevant until today [22, 34, 57]. In recent years, metamorphic testing has gained popularity as an alternative [82, 83, 91]. There is some work on generating test cases to maximize coverage [115].

One of the oldest areas where the input of a structured language is required is *compiler testing* [18]. Similarly to generating knowledge graphs as test inputs, it is important to generate test cases with a *semantic difference*, not only test cases that are syntactically different. Another similarity is that compilers, like OWL reasoners, often contain a lot of complex optimizations, which make the tool faster but can be a source of bugs. Many approaches to compiler testing are based on the grammar of the input language [12, 40, 62, 78, 88, 111] but mutation-based fuzzing is also a common, more recent alternative [25,33,61,69]. Differential testing is a common way to build the oracle, where there is not only work on comparing between different compilers [87, 96], but also between different versions of the same compiler [42, 96] and between different optimizations of the same compiler [8, 70]. Again, metamorphic testing is a common alternative [60, 85, 98].

A third area of interest is the fuzzing of *SMT solvers.* Similar to OWL reasoners, these tools perform a logical analysis where trust in the result is absolutely crucial. They also form the basis of many applications. Fuzzing SMT solvers is a rather recent field that helped to uncover dozens of bugs in state-of-the-art solvers, justifying the need for such testing methods. Fuzzing of SMT solvers is either grammar-based [14, 105] or mutation-based [65]. Again, methods for differential testing [14, 105] and metamorphic testing [65, 106] are used.

3 Testing Knowledge Graph Applications

Classifying Knowledge Graph Applications. There are numerous applications operating on knowledge graphs and ontologies, and as the concrete testing strategy depends on the program, we distinguish three classes of software. In the following, we focus on the RDF/OWL technology stack.

Generic Tools: Generic tools take as input any knowledge graph or ontology as defined by the RDF or OWL standards for input syntaxes. APIs such as the OWL-API [47] and frameworks such as Apache Jena fall under this class, including their parsers and generic functions. Similarly, editors such as Protégé [68] fall under this class.

Specialized Tools: Specialized tools take as input any knowledge graph or ontology in an OWL profile or RDF that is defined explicitly and outside the tool. For example, EL reasoners take as input any ontology in the OWL EL profile, which is specified as part of the standard. Similarly, knowledge graph construction tools may assume that the input confirms to a certain vocabulary [21], and SHACL shapes may be employed to enforce this assumption. Such tools build on generic tools.

Integrated Applications: An integrated application is a program where the knowledge graph or ontology is not the main input, but an auxiliary input. Testing integrated applications is most complex for both input generation and testing oracle. The input is a subset of RDF/OWL that is not explicitly defined and explicit testing oracles require knowledge of the application.

We conduct two fuzzing campaigns, one for generic tools, one for specialized tools, namely EL reasoners. For integrated applications, we refer to John et al. [55], where several case studies illustrate the challenges and possible solutions.

Campaign 1: RDF-TTL. Campaign 1 focuses on generic tools, namely the RDF Turtle parsers of the OWL-API and Apache Jena, the most widely used open source tools. As the basis for the ISLa specification, we use the grammar and restrictions for RDF 1.1 Turtle, which are provided in the corresponding standard [16, Sec. 6.5]. The campaign uses the implicit crash oracle to check whether we can crash the parser.

Campaign 2: OWL EL. Campaign 2 focuses on specialized tools, namely the three EL reasoners bundled with Protégé, which is the most widely used editor: HermiT [35], ELK [59] and Pellet [89]. EL reasoners are assumed to be stable [74], so we assume that less blocking bugs will occur by restricting the input ontologies to the OWL EL profile. As input, we generate random ontologies in the OWL EL profile, using the functional syntax of OWL, and with a restricted number of IRIs. As the basis for the ISLa specification, we use the grammar for ontologies in the OWL EL profile, which is provided in the corresponding standard [48, Sec. 2, Sec. 6]. Figure 1 shows a simplified excerpt of the grammar that we provided to ISLa. We included all non-terminals described in the grammar from the standard and thus are able to produce all structures of ontologies in the OWL EL profile.

```
<ontology>       ::= "Ontology (" <declarations> " " <axioms> ")"
<axioms>         ::= <axiom> | <axiom> "\n" <axioms>
<axiom>          ::= <classAxiom> | <assertion> | <dataTypeDefinition> | [...]
<classAxiom>     ::= <subClassOf> | <equivClasses> | <disjointClasses> | [...]
<equivClasses>   ::= "EquivalentClasses(" <classExpr> " " <classExpr> ")"
<classExpr>      ::= <Class> | <objectOneOf> | <dataHasValue> | [...]
<objectOneOf>    ::= "ObjectOneOf(" <Individual> ")"
<dataHasValue>   ::= "DataHasValue(" <ObjectProperty> " " <literal> ")"
<literal>        ::= <typedLiteral> | <stringNoLang> | <stringWithLang>
<stringNoLang>   ::= <QuotedString>
<stringWithLang> ::= <QuotedString> <LanguageTag>
```

Fig. 1. Excerpt from the grammar for our RDF-TTL campaign.

We employ two differential testing oracles: (1) we ask all reasoners to check the input for consistency, and (2) we ask all reasoners to derive all possible axioms and check whether they derive the same.

If any of the reasoners answers differently, we investigate manually which answer is wrong. As an additional target, we also use the ontology as input to the classifier of the OWL-API to check whether the API correctly detects that it is in the OWL EL profile.

4 Application

In the following, we describe the two testing campaigns in detail. The test case generators for the two testing campaigns are implemented based on ISLa. The oracle for both campaigns, which calls the different system-under-test and logs anomalies, is implemented using Java. The implementation, including the ISLa grammars and constraints, the test cases, logged anomalies and produced bug reports, is available online [56]. During the setup of the testing pipeline, we run the initial experiments on a personal computer with an Intel i7-1165G7 CPU @ 2.80GHz running Ubuntu 22.04 with a RAM limit of 8GB. The main test runs are performed on a server with an Intel Xeon Gold 6240 CPU @ 2.60GHz running Ubuntu 22.04 with a RAM limit of 4GB.

4.1 RDF-TTL Testing Campaign

Test Case Generator. To set up the test case generator for the RDF 1.1 Turtle file format [16], we slightly modify the grammar: First, we exclude SPARQL-style prefix definitions, as this triggers an already known bug [72, issue 1149], which would be a blocking bug. Secondly, we restrict the grammar to use a fixed set of literals, i.e. IRIs, numbers, strings, blank node labels, language tags and prefixes. This is to enforce that the same symbols are used in several axioms, as ISLa otherwise generates inputs where each symbol is used once. Thirdly, we modify the starting rule to generate at least ten statements and add all prefix definitions in the beginning. This is to avoid a discovered issue during initial testing (see below).

Systems-under-Test and Procedure. We test the Turtle parser of the OWL-API (version 5.5.0) and the Turtle parser of Apache Jena (version 5.1.0). We use the most recent versions of the tools at the time when the testing started. Already while specifying the grammar, we identified an imprecision in the current RDF Turtle standard that impacted the design of our test case generator. Initial testing of the setup on a laptop revealed five issues, and we subsequently restricted the grammar to not use the features that trigger some found anomalies. Finally, we performed the main test run for 24 hours, producing and evaluating 27,665 test cases. This larger test run did not reveal additional anomalies. Thus, we found anomalies resulting in five bug reports.

4.2 OWL EL Testing Campaign

Test Case Generator. We make similar adjustments to the grammar as before. Firstly, we again restrict the grammar to use four different names for classes, individuals, simple object properties, potentially non-simple object properties, data properties, predefined data types, custom data types, language tags, strings, annotation properties and annotation values, respectively. Secondly, the grammar is modified to encode some of the constraints on the EL profile that are not contained in the original grammar, e.g., the restrictions on complex roles and the restriction on the minimal number of arguments for HasKey axioms. Initial testing showed that the encoding in the grammar leads to faster generation times than providing the constraints to ISLa. We do not encode all of the constraints, namely that cyclic definitions of properties are forbidden and the *global restrictions on the EL profile* [48]. Thus, we also generated test cases that are not in OWL EL, which the profile checker and reasoners are expected to reject.

Systems under Test. We test the reasoners HermiT (v.1.4.5.519), Pellet/Openllet (v.2.6.5), ELK (v.0.6.0) and the EL-profile checker of the OWL-API (v.5.5.0). We use the most recent versions of the tools at the time when the testing started. We test two capabilities: checking the consistency of an ontology, and inferring all axioms of the following types: SubClass, DisjointClasses, EquivalentClasses, ClassAssertion, PropertyAssertion, SubObjectProp, SubDataProp, EquivalentObjectProp, EquivalentDataProp, as well as characteristics of object and data properties, e.g. functionaliy or symmetry. As the ELK reasoner does not support the inference of all of these types of axioms, the oracle checks if the inferred axioms of ELK are a subset of the axioms inferred by the other reasoners.

Test Procedure. Initial test runs already revealed some anomalies and for the main test run, the oracle for the EL-profile checker is adjusted to not identify the same anomalies again. The main test run was performed for 10 hours, producing and evaluating 1,557 test cases. Out of those, 516 test cases contain at

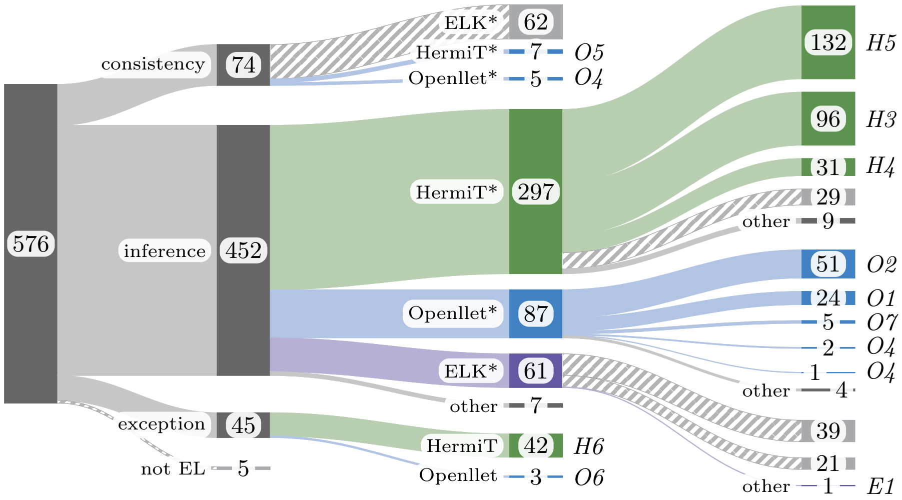

Fig. 2. Classification of 576 found anomalies. Legend: * = result of this tool is outlier, ▨ = anomaly is not a bug but due to limitations of the tools, italic labels = IDs from Table 2 for bug in this class.

least one anomaly. The test cases are manually classified according to which tools' output does not conform to the output of the other tools and patterns in the anomalies.Because of the high number of anomalies, not all anomalies were investigated individually, instead we examined at least one test case in each class.

Some classes turn out to not be bugs of the reasoners but rather mirror specific reasoning limitations of the reasoners. We do not examine unclassified anomalies with very different behavior of the tools as this indicates the occurrence of several anomalies at the same time, which we already examine individually in other classes. While investigating the anomalies, the input files are simplified, i.e. axioms and parts of axioms that do not affect the result are removed. Classes, individuals and properties are renamed to make the files easier to understand. This last step can be seen as a form of metamorphic testing and indeed revealed one of the bugs, which is discussed in detail in Section 4.3.

Found Anomalies. The initial test runs revealed eight anomalies in Openllet, Hermit and the EL-profile checker. The main test run resulted in 16 classes containing 576 anomalies. Out of those, 420 are proper anomalies, i.e. anomalies not resulting from tool limitations. An overview of the classification of the anomalies is depicted in Figure 2. Investigating representatives of the different classes of anomalies lead to 15 newly discovered bugs in total.

4.3 Found Bugs

Overall, our testing methodology revealed 21 previously unknown bugs. Tables 1 and 2 provide an overview of these bugs. We found bugs in all systems that we

Table 1. All bugs found in the RDF-TTL campaign. The issueIds refer to the corresponding trackers for the OWL-API [72], Apache Jena [53] and RDF [81] Legend: S = Soundness, D = Documentation, ○ = confirmed, but not fixed. Only bug *A4* is logical.

Tool/Std.	ID	IssueId	Type	Fixed?	Summary
OWL-API	*A4*	1155	S		Value for empty prefix set without definition
	A5	1151	S	○	Empty list not always parsed correctly
	A6	1152	D	✓	Raised warning too unspecific
Apache Jena	*J1*	2715	S	✓	Parsing some floating point numbers fails
RDF 1.1/1.2	*R1*	59	D	✓	Placing of prefix definitions not restricted

tested. In particular, all tested reasoners contain at least one of the found bugs. We reported the bugs in 16 issues in the repositories of the tested tools, except for HermiT where no bug tracker is publicly available. The detailed reports for those bugs (as well as all other bugs) can be found in our supplementary material [56]. One interesting observation is that most of the found bugs are *logical bugs*, i.e. bugs that do not lead to any exception or warning but the computed result is incorrect. Logical bugs are particularly important to find, as there is no indication of their presence for users of the tools. To illustrate the kinds of bugs we found, we discuss a few different classes of bugs using examples.

Imprecision of Standard. One bug (*R1* in Table 1) was already discovered when formalizing the grammar and constraints of the Turtle format. It is an underspecification of the RDF 1.1 Turtle Standard that was also present in the working draft for RDF 1.2 Turtle [101] at the time we performed the testing. The standard does not define whether used prefixes have to be defined before they are used: It states that the prefix has to be defined *outside* its use, instead of *before.* Due to our report, the working draft for RDF 1.2 Turtle has been clarified and such input files are not allowed [100].

Parsing Files. Six of the discovered bugs occur in the tested parsers. One example is shown in Figure 3 (1), which corresponds to *A5* in Table 1. The file can not be parsed due to the second line, which contains an empty relation for `<iri1>`. Instead, an exception is thrown. A second example is shown in Figure 3 (2), which corresponds to *J1* in Table 1. The file can not be parsed by Apache Jena: The combination of a sign followed by a dot, without having a leading zero, is allowed according to the standard but could not be parsed by Apache Jena. Instead, an exception is thrown.

Parser-related bugs were also found in the EL-checker of the OWL-API: While the OWL EL profile allows the use of language tags, the parser combines strings and attached language tags into one objects of type `rdfs:langString`. The EL-check subsequently rejects the ontology because of the presence of this data type, effectively forbidding the use of language tags.

Table 2. Bugs from the OWL EL campaign. **ID** refers to labels in Fig. 2. **IssueId** refers to the corresponding issue trackers [27,71,72]. The bugs for HermiT can be found in our supplementary material [56]. *O1*, *H1*, *H2*, *A1*, *A2* and *A3* were discovered during initial testing (see Fig. 2). All but *O6* and *H6* are logical bugs. Legend: S = Soundness, SC = Soundness (consistency), SI = Soundness (inference), C = Completeness, D = Documentation, E = Exception.

Tool	ID	IssueId	Task	Type	Summary
Pellet	*O1*	85	Reasoner	SC	Equivalence to bottom property is missing when range of property is empty
	O2	87	Reasoner	SI	Incorrectly infers asymmetry of property from a single triple
	O3	88	Reasoner	SI	Incorrect equivalence with bottom class
	O4	89	Reasoner	S	Renaming of variables leads to change in consistency check
	O5	90	Reasoner	SC	Ignores language tags when checking for equivalence of literals
	O6	91	Reasoner	E	Null-pointer exception on invocation
	O7	92	Reasoner	SI	Incorrectly infers irreflexivity of property from single triple
ELK	*E1*	71	Reasoner	SC	Ignores language tags when considering equivalence of literals.
HermiT	*H1*	—	Reasoner	C	Missing `owl:Thing` class assertion of declared individual when no class is declared
	H2	—	Reasoner	D	Warning about malformed input missing and computed result has SI issue
	H3	—	Reasoner	C	Functionality of property missing when property range is a singleton
	H4	—	Reasoner	SI	Incorrect sub-object-property relation caused by unrelated subclass axiom
	H5	—	Reasoner	C	Data-property assertion missing when using singleton in equivalent-classes axiom
	H6	—	Reasoner	E	Exception when using only `rdfs:Literal` in data-type intersection
OWL-API	*A1*	1158	Parser	S	Forbidden data type is wrongly detected because of incorrect parsing of string with language tag as `rdfs:langString`
	A2	1159	Parser	D	Too few arguments are detected because the list of arguments is interpreted as set
	A3	1160	EL-Checker	S	Defining new data type is incorrectly marked as violation (only use is forbidden)

Consistency. Four bugs are cases where the reasoners do not correctly assess the consistency of the ontology. We found three such bugs in Pellet/Openllet and one in ELK. The latter (*E1* in Table 2) is shown in Figure 4. The ontology is wrongly classified as inconsistent by ELK: The root cause for the bug is again an

```
<iri1> <iri2> <iri3> ; ; .
```

(bug *A5*)

```
<iri1> <iri2> +.7 .
```

(bug *J1*)

Fig. 3. Turtle files that trigger bugs in parsers: (*A5*) OWL-API, (*J1*) Apache Jena

```
Prefix(:=<http://www.example.org/reasonerTester#>)
Ontology (
    Declaration(Class(:B))    Declaration(Class(:A))
    Declaration(DataProperty(:dr))    Declaration(NamedIndividual(:a))
    EquivalentClasses( DataHasValue(:dr "s1"@fr)   :A   :B )
    DisjointClasses(   DataHasValue(:dr "s1"@en)   :A )
    ClassAssertion(:B :a))
```

Fig. 4. Consistency bug in ELK (bug *E1*)

insufficient treatment of the language tags. ELK treats the two strings `"s1"@fr` and `"s1"@en` as the same entity, even though they must be treated as different literals. This leads to the incorrect inference that `:B` is an empty class and, due to the class assertion **ClassAssertion**`(:B :a)`, that the ontology is inconsistent.

One consistency bug for Openllet/Pellet (*O5*) is shown in Figure 5. The ontology contains a functional data property `:dp`. Hence, the intersection of the two classes described with DataHasValue is empty and therefore both cannot be equivalent to `owl:Thing`. Despite this, Openllet classifies the ontology incorrectly as consistent. The consistency assessment is correct, if one removes the language tag `@en`. Therefore, this bug is also due to incorrect handling of language tags.

Soundness of Inference. Five of the bugs are cases where the reasoners infer axioms that are not entailed by the ontology. These bugs were found in HermiT and Pellet/Openllet. Figure 6 shows one occurrence of such a bug for Pellet/Openllet, which corresponds to *O4* in Table 2. The ontology contains only one axiom and declares an individual `:a` that is not mentioned in the axiom. Nevertheless, it is inferred that the axiom **ClassAssertion**`(:C :a)` is entailed by the ontology. Interestingly, changing the name of the declared class, the individuals or the properties leads to a correct inference, i.e., the bug disappears. We see this puzzling behavior of Pellet/Openllet for several different inputs, some of which lead to an incorrect consistency assessment.

```
Prefix(:=<http://www.example.org/reasonerTester#>)
Ontology (
  Declaration(DataProperty(:dp)) FunctionalDataProperty(:dp)
  EquivalentClasses(owl:Thing DataHasValue(:dp "s1"@en) DataHasValue(:dp "s2")))
```

Fig. 5. Consistency bug in Pellet/Openllet (bug *O5*)

```
Prefix(:=<http://www.example.org/reasonerTester#>)
Ontology (
    Declaration(Class(:C))                  Declaration(ObjectProperty(:qsim))
    Declaration(NamedIndividual(:a))        Declaration(NamedIndividual(:d))
    EquivalentClasses( ObjectHasSelf(:qsim) ObjectOneOf(:d) :C ))
```

Fig. 6. Inference/soundness bug in Pellet/Openllet (bug *O4*)

```
Prefix(:=<http://www.example.org/reasonerTester#>)
Ontology (
    Declaration(DataProperty(:dp))  Declaration(NamedIndividual(:a))
    EquivalentClasses( ObjectOneOf(:a)  DataHasValue(:dp "data") ))
```

Fig. 7. Inference bug: completeness bug in HermiT (bug *H5*)

Completeness of Inference. Three of the bugs are cases where the reasoners do not infer an axiom although it is entailed by the ontology. All these bugs were found in HermiT. Figure 7 shows one input, which corresponds to *H5* in Table 2. The ontology defines a data property `:dp` and an individual `:a`. Furthermore, two classes are the same: (i) the class containing only the individual `:a` and (ii) the class of individuals occurring as the subject in an assertion with the data property `:dp` and the object `"data"`. One can therefore infer the assertion **DataPropertyAssertion**`(:dp :a "data")` from the ontology. However, HermiT does not infer the axiom when asked to compute all inferred assertions.

5 Discussion

Effects in Bug Finding. As to be expected, we found bugs in parts of the grammars which are less often used in general, in particular language tags. However, it was not expected that several issues are related to different interpretations of the standard (cf. *R1*) or are part of the field folklore but are not documented in an obvious place (cf. *A2* the issue that the grammar defines arguments as lists of parameters, but the parser silently converts them into a set).

Several issues were related to documentation, where the error report was due to incorrect usage of the tool, but did not point out what exactly the problem is (cf. *H6*). In fact, unclear error messages are a common problem that goes deeper than wording, but also touches on exposure of concepts [99], and we see these results as a first indication that this is also the case for knowledge graphs.

One might assume that unsound inferences would always lead to wrong consistency classifications, but changing the input to detect such a consistency bug failed for HermiT. When we added axioms to achieve inconsistency, the set of inferred axioms did indeed become inconsistent, but when asked for consistency of the ontology the result was still correct. We did succeed constructing a consistency bug for ELK (*E1*) from an unsound inference. This indicates that further testing efforts for reasoners will require more in-depth knowledge about internals.

Limitations. While we discovered bugs in all involved tools, and uncovered an imprecision in the RDF-TTL 1.1 standard, our work does not give an overview over the state of software quality for knowledge graphs or ontologies. Neither did we consider commercial tools or conducted a systematic approach to selection of the tested software, nor did we cover all functions of the tools that we selected. Similarly, the oracles are kept simple. Our aim is to demonstrate that software quality in applications on knowledge graphs and ontologies can be improved using automated testing with acceptable effort to do so, and the found bugs and ease of setup confirm that this is indeed the case.

Actionable Insights. The first insight gained from our study is that automated testing with derived oracles can be setup with little effort, thus increasing the quality of the tools in knowledge graph and ontology research. For the generic and specialized tool classes, grammars are reusable; for example, any tool operating on EL ontologies can build on the grammar and constraints presented here. Implicit oracles do not require any setup, and differential testing is easily applicable whenever a new software is evaluated against other tools. Thus, even academic software, often suffering from quality and maintenance problems, can be improved without much training in testing or software development. Furthermore, we see a great potential in metamorphic testing, especially for reasoners. As one of the bugs above witnesses, renaming variables can lead to differences in the classification, but closure under renaming of variables or reordering of axioms is a property that should be assumed for any reasoner.

The second insight is that fuzzing can be used to increase the overall quality of software in the field, thus increasing the confidence in the tools and avoiding compromising the results of studies that build on them. For specialized applications and specified test oracles, developers must be involved, but such campaigns can be conducted by others to lessen the load of work on the developers.

6 Conclusion

Software reliability, or software quality in general, is an underestimated and underappreciated factor when it comes to knowledge graphs, yet reliability is critical for the acceptance of this technology. This work is the first systematic bug finding study for this class of software and reports on bugs in all considered tools and imprecisions in standards and documentation. This demonstrates that automated testing has reached a state where it can be applied to graph data. Based on our actionable insights and discussion of the specific challenges of bugs in presence of reasoning, we hope that the quality of the software tailored to handling ontologies and knowledge graphs can be increased. For future work, it remains to investigate further testing techniques, in particular greybox fuzzing [10] and other approaches that are building on *coverage* to estimate how much of the program has been already tested.

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

References

1. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Reviews of modern physics **74**(1), 47 (2002)
2. Alvaro, P., Rigger, M.: Automatically testing database systems: DBMS testing with test oracles, transaction history, and fuzzing. ACM Queue **21**(6), 128–135 (2024)
3. Anand, S., Burke, E.K., Chen, T.Y., Clark, J.A., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. **86**(8), 1978–2001 (2013)
4. Armstrong, T.G., Ponnekanti, V., Borthakur, D., Callaghan, M.: LinkBench: a database benchmark based on the Facebook social graph. In: International Conference on Management of Data (SIGMOD Conference). pp. 1185–1196. ACM (2013). `https://doi.org/10.1145/2463676.2465296`, `https://doi.org/10.1145/2463676.2465296`
5. Ba, J., Rigger, M.: Keep it simple: Testing databases via differential query plans. Proc. ACM Manag. Data **2**(3), 188 (2024)
6. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H.L., Lemay, A., Advokaat, N.: gMark: Schema-driven generation of graphs and queries. IEEE Trans. Knowl. Data Eng. **29**(4), 856–869 (2017). `https://doi.org/10.1109/TKDE.2016.2633993`, `https://doi.org/10.1109/TKDE.2016.2633993`
7. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Trans. Software Eng. **41**(5), 507–525 (2015)
8. Béra, C., Miranda, E., Denker, M., Ducasse, S.: Practical validation of bytecode to bytecode JIT compiler dynamic deoptimization. J. Object Technol. **15**(2), 1:1–26 (2016). `https://doi.org/10.5381/JOT.2016.15.2.A1`
9. Binnig, C., Kossmann, D., Lo, E., Özsu, M.T.: QAGen: Generating query-aware test databases. In: International Conference on Management of Data (SIGMOD Conference). pp. 341–352. ACM (2007). `https://doi.org/10.1145`
10. Böhme, M., Pham, V., Nguyen, M., Roychoudhury, A.: Directed greybox fuzzing. In: Conference on Computer and Communications Security (CCS'17). pp. 2329–2344. ACM (2017)
11. Böhme, M., Pham, V., Roychoudhury, A.: Coverage-Based Greybox Fuzzing as Markov Chain. IEEE Trans. Software Eng. **45**(5), 489–506 (2019). `https://doi.org/10.1109/TSE.2017.2785841`
12. Boujarwah, A.S., Saleh, K., Al-Dallal, J.: Testing syntax and semantic coverage of Java language compilers. Inf. Softw. Technol. **41**(1), 15–28 (1999). `https://doi.org/10.1016/S0950-5849(98)00075-5`
13. Bringolf, M., Winterer, D., Su, Z.: Finding and understanding incompleteness bugs in SMT solvers. In: International Conference on Automated Software Engineering (ASE). pp. 43:1–43:10. ACM (2022)
14. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories. pp. 1–5 (2009)
15. 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`

16. Carothers, G., Prud'hommeaux, E.: RDF 1.1 Turtle. W3C recommendation, W3C (Feb 2014), https://www.w3.org/TR/2014/REC-turtle-20140225/
17. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: International Conference on Data Mining. pp. 442–446. SIAM (2004). https://doi.org/10.1137/1.9781611972740.43
18. Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L.: A Survey of Compiler Testing. ACM Computing Surveys **53**(1), 1–36 (Jan 2021). https://doi.org/10.1145/3363562
19. 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
20. Collarana, D., Galkin, M., Lange, C., Scerri, S., Auer, S., Vidal, M.: Synthesizing knowledge graphs from web sources with the MINTE$^+$ framework. In: International Semantic Web Conference (ISWC 2018) , Part II. LNCS, vol. 11137, pp. 359–375. Springer (2018). https://doi.org/10.1007/978-3-030-00668-6_22
21. Conde-Herreros, D., Stork, L., Pernisch, R., Poveda-Villalón, M., Corcho, Ó., Chaves-Fraga, D.: Propagating ontology changes to declarative mappings in construction of knowledge graphs. In: International Workshop on Knowledge Graph Construction (KGCW@ESWC). CEUR Workshop Proceedings, vol. 3718. CEUR-WS.org (2024)
22. Cui, Z., Dou, W., Dai, Q., Song, J., Wang, W., Wei, J., Ye, D.: Differentially Testing Database Transactions for Fun and Profit. In: International Conference on Automated Software Engineering (ASE). pp. 35:1–35:12 (2022). https://doi.org/10.1145/3551349.3556924
23. de la Riva, C., Suárez-Cabal, M.J., Tuya, J.: Constraint-based test database generation for SQL queries. In: Workshop on Automation of Software Test. pp. 67–74. ACM (2010). https://doi.org/10.1145/1808266.1808276
24. Demeyer, S., Roover, C.D., Beyazit, M., Härtel, J.: Threats to instrument validity within "in silico" research: Software engineering to the rescue. In: Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering Methodologies ISoLA (4). LNCS, vol. 15222, pp. 82–96. Springer (2024)
25. Donaldson, A.F., Evrard, H., Lascu, A., Thomson, P.: Automated testing of graphics shader compilers. Proc. ACM Program. Lang. **1**(OOPSLA), 93:1–93:29 (2017). https://doi.org/10.1145/3133917
26. Dong, X.L.: Generations of knowledge graphs: The crazy ideas and the business impact. Proc. VLDB Endow. **16**(12), 4130–4137 (2023)
27. Elk reasoner, issue tracker, https://github.com/liveontologies/elk-reasoner/issues
28. Erdős, P., Rényi, A.: On random graphs I. Publ. math. debrecen **6**(290-297), 18 (1959)
29. 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 (GDD^{X}s). IEEE Access **9**, 5609–5639 (2021)
30. 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
31. Foundation, A.S.: Apache Jena, available at: https://jena.apache.org/, accessed 19-July-2008

32. Gao, W., Pham, V., Liu, D., Chang, O., Murray, T., Rubinstein, B.I.P.: Beyond the Coverage Plateau: A Comprehensive Study of Fuzz Blockers (Registered Report). In: International Fuzzing Workshop (FUZZING). pp. 47–55. ACM (2023). https://doi.org/10.1145/3605157.3605177
33. Garoche, P., Howar, F., Kahsai, T., Thirioux, X.: Testing-based compiler validation for synchronous languages. In: Symposium on NASA Formal Methods (NFM). LNCS, vol. 8430, pp. 246–251. Springer (2014). https://doi.org/10.1007/978-3-319-06200-6_19
34. Ghit, B., Poggi, N., Rosen, J., Xin, R., Boncz, P.A.: SparkFuzz: Searching correctness regressions in modern query engines. In: International Workshop on Testing Database Systems (DBTest@SIGMOD 2020). pp. 1:1–1:6 (2020). https://doi.org/10.1145/3395032.3395327
35. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: Hermit: An OWL 2 reasoner. J. Autom. Reason. **53**(3), 245–269 (2014)
36. 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
37. 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
38. Gottschalk, S., Demidova, E.: *Tab2KG*: Semantic table interpretation with lightweight semantic profiles. Semantic Web **13**(3), 571–597 (2022). https://doi.org/10.3233/SW-222993
39. Gutierrez, C., Sequeda, J.F.: Knowledge graphs. Commun. ACM **64**(3), 96–104 (2021)
40. Hanford, K.V.: Automatic generation of test cases. IBM Syst. J. **9**(4), 242–257 (1970). https://doi.org/10.1147/SJ.94.0242
41. Havrikov, N.: Grammar-based fuzzing using input features. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2021), https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/32722
42. Hawblitzel, C., Lahiri, S.K., Pawar, K., Hashmi, H., Gokbulut, S., Fernando, L., Detlefs, D., Wadsworth, S.: Will you still compile me tomorrow? static cross-version compiler validation. In: Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering, (ESEC/FSE). pp. 191–201. ACM (2013). https://doi.org/10.1145/2491411.2491442
43. Hitzler, P.: A review of the semantic web field. Commun. ACM **64**(2), 76–83 (2021). https://doi.org/10.1145/3397512
44. Hodován, R., Kiss, Á., Gyimóthy, T.: Grammarinator: a grammar-based open source fuzzer. In: International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST@SIGSOFT FSE. pp. 45–48. ACM (2018). https://doi.org/10.1145/3278186.3278193
45. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., Ngomo, A.N., Polleres, A., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J.F., Staab, S., Zimmermann, A.: Knowledge graphs. ACM Comput. Surv. **54**(4), 71:1–71:37 (2022)
46. Holler, C., Herzig, K., Zeller, A.: Fuzzing with Code Fragments. In: USENIX Security Symposium. pp. 445–458. USENIX Association (2012), https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

47. Horridge, M., Bechhofer, S.: The OWL API: A Java API for working with OWL 2 ontologies. In: International Workshop on OWL: Experiences and Directions OWLED. CEUR Workshop Proceedings, vol. 529. CEUR-WS.org (2009)
48. 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
49. Houkjær, K., Torp, K., Wind, R.: Simple and realistic data generation. In: International Conference on Very Large Data Bases (VLDB). pp. 1243–1246 (2006)
50. Hua, Z., Lin, W., Ren, L., Li, Z., Zhang, L., Jiao, W., Xie, T.: GDsmith: Detecting bugs in cypher graph database engines. In: International Symposium on Software Testing and Analysis (ISSTA). pp. 163–174. ACM (2023). https://doi.org/10.1145/3597926.3598046
51. Hubert, N., Monnin, P., d'Aquin, M., Monticolo, D., Brun, A.: Pygraft: Configurable generation of synthetic schemas and knowledge graphs at your fingertips. In: The Semantic Web - International Conference ESWC (2). LNCS, vol. 14665, pp. 3–20. Springer (2024)
52. Ileri, A.M., McGinty, H.: VEL: A formally verified reasoner for EL++ description logic. In: International Semantic Web Conference ISWC (Posters/Demos/Industry). International Workshop on OWL: Experiences and Directions, CEUR Workshop Proceedings, vol. 3828. CEUR-WS.org (2024)
53. Apache Jena, issue tracker, https://github.com/apache/jena/issues
54. Jiang, Y., Liu, J., Ba, J., Yap, R.H.C., Liang, Z., Rigger, M.: Detecting logic bugs in graph database management systems via injective and surjective graph query transformation. In: International Conference on Software Engineering (ICSE). pp. 46:1–46:12. ACM (2024). https://doi.org/10.1145/3597503.3623307
55. John, T., Johnsen, E.B., Kamburjan, E.: Mutation-based integration testing of knowledge graph applications. In: International Symposium on Software Reliability Engineering (ISSRE). pp. 475–486. ACM (2024). https://doi.org/10.1109/ISSRE62328.2024.00052
56. John, T., Johnsen, E.B., Kamburjan, E., Steinhöfel, D.: Supplementary material for paper "Language-based testing for knowledge graphs", Zenodo (Jan 2025). https://doi.org/10.5281/zenodo.14512591
57. Jung, J., Hu, H., Arulraj, J., Kim, T., Kang, W.H.: APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. Proc. VLDB Endow. **13**(1), 57–70 (2019)
58. Kamm, M., Rigger, M., Zhang, C., Su, Z.: Testing Graph Database Engines via Query Partitioning. In: International Symposium on Software Testing and Analysis (ISSTA). pp. 140–149. ACM (2024). https://doi.org/10.1145/3597926.3598044
59. Kazakov, Y., Krötzsch, M., Simancik, F.: The incredible ELK - from polynomial procedures to efficient reasoning with EL ontologies. J. Autom. Reason. **53**(1), 1–61 (2014)
60. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: Conference on Programming Language Design and Implementation (PLDI). pp. 216–226. ACM (2014). https://doi.org/10.1145/2594291.2594334
61. Le, V., Sun, C., Su, Z.: Finding deep compiler bugs via guided stochastic program mutation. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015). pp. 386–399. ACM (2015). https://doi.org/10.1145/2814270.2814319
62. Lindig, C.: Random testing of C calling conventions. In: International Workshop on Automated Debugging (AADEBUG). pp. 3–12. ACM (2005). https://doi.org/10.1145/1085130.1085132

63. Liu, S., Lan, J., Du, X., Li, J., Lu, W., Jiang, J., Du, X.: Testing Graph Database Systems with Graph-State Persistence Oracle. In: International Symposium on Software Testing and Analysis (ISSTA). pp. 666–677. ACM (2024). https://doi.org/10.1145/3650212.3680311
64. Lu, Y., Hou, W., Pan, M., Li, X., Su, Z.: Understanding and finding Java decompiler bugs. Proc. ACM Program. Lang. **8**(OOPSLA1), 1380–1406 (2024)
65. Mansur, M.N., Christakis, M., Wüstholz, V., Zhang, F.: Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 701–712. ACM (2020). https://doi.org/10.1145/3368089.3409763
66. 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
67. Miller, G.: A scientist's nightmare: Software problem leads to five retractions. Science **314**(5807), 1856–1857 (2006). https://doi.org/10.1126/science.314.5807.1856, https://www.science.org/doi/abs/10.1126/science.314.5807.1856
68. Musen, M.A.: The Protégé project: A look back and a look forward. AI Matters **1**(4), 4–12 (2015). https://doi.org/10.1145/2757001.2757003
69. Nagai, E., Hashimoto, A., Ishiura, N.: Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. IPSJ Trans. Syst. LSI Des. Methodol. **7**, 91–100 (2014). https://doi.org/10.2197/IPSJTSLDM.7.91
70. Ofenbeck, G., Rompf, T., Püschel, M.: Randir: differential testing for embedded compilers. In: Symposium on Scala (SCALA@SPLASH). pp. 21–30. ACM (2016). https://doi.org/10.1145/2998392.2998397
71. Openllet, issue tracker, https://github.com/Galigator/openllet/issues
72. OWL-API, issue tracker, https://github.com/owlcs/owlapi/issues
73. Park, H., Kim, M.: Trilliong: A trillion-scale synthetic graph generator using a recursive vector model. In: International Conference on Management of Data (SIGMOD Conference). pp. 913–928. ACM (2017). https://doi.org/10.1145/3035918.3064014
74. Parsia, B., Matentzoglu, N., Gonçalves, R.S., Glimm, B., Steigmiller, A.: The OWL reasoner evaluation (ORE) 2015 competition report. J. Autom. Reason. **59**(4), 455–482 (2017). https://doi.org/10.1007/S10817-017-9406-8
75. 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 (Dec 2017). https://doi.org/10.1007/s10817-017-9406-8
76. Pham, M., Boncz, P.A., Erling, O.: S3G2: A scalable structure-correlated social graph generator. In: Performance Evaluation and Benchmarking: TPC Technology Conference (TPCTC). LNCS, vol. 7755, pp. 156–172. Springer (2012). https://doi.org/10.1007/978-3-642-36727-4_11
77. Portisch, J., Paulheim, H.: The DLCC node classification benchmark for analyzing knowledge graph embeddings. In: International Semantic Web Conference (ISWC 2022). LNCS, vol. 13489, pp. 592–609. Springer (2022). https://doi.org/10.1007/978-3-031-19433-7_34
78. Purdom, P.: A sentence generator for testing parsers. BIT Numerical Mathematics **12**, 366–375 (1972)

79. Püroja, D., Waudby, J., Boncz, P.A., Szárnyas, G.: The LDBC social network benchmark interactive workload v2: A transactional graph query benchmark with deep delete operations. In: Performance Evaluation and Benchmarking: TPC Technology Conference (TPCTC). LNCS, vol. 14247, pp. 107–123. Springer (2023). https://doi.org/10.1007/978-3-031-68031-1_8
80. Raynaud, T., Amir, S., Haque, R.: A generic and high-performance RDF instance generator. Int. J. Web Eng. Technol. **11**(2), 133–152 (2016). https://doi.org/10.1504/IJWET.2016.077342
81. RDF 1.2 Turtle standard, issue tracker, https://github.com/w3c/rdf-turtle/issues
82. Rigger, M., Su, Z.: Finding bugs in database systems via query partitioning. Proc. ACM Program. Lang. **4**(OOPSLA), 211:1–211:30 (2020). https://doi.org/10.1145/3428279
83. Rigger, M., Su, Z.: Testing Database Engines via Pivoted Query Synthesis. In: Symposium on Operating Systems Design and Implementation (OSDI). pp. 667–682 (2020)
84. Samanta, B., De, A., Jana, G., Gómez, V., Chattaraj, P.K., Ganguly, N., Gomez-Rodriguez, M.: NEVAE: A deep generative model for molecular graphs. J. Mach. Learn. Res. **21**, 114:1–114:33 (2020), https://jmlr.org/papers/v21/19-671.html
85. Samet, H.: A normal form for compiler testing. In: Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages. pp. 155–162. ACM (1977). https://doi.org/10.1145/800228.806945
86. 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
87. Sheridan, F.: Practical testing of a C99 compiler using output comparison. Softw. Pract. Exp. **37**(14), 1475–1488 (2007). https://doi.org/10.1002/SPE.812
88. Sirer, E.G., Bershad, B.N.: Using production grammars in software testing. In: Conference on Domain-Specific Languages (DSL). pp. 1–13. ACM (1999). https://doi.org/10.1145/331960.331965
89. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. Web Semant. **5**(2), 51–53007 (2007)
90. Slutz, D.R.: Massive stochastic testing of SQL. In: International Conference on Very Large Data Bases (VLDB). vol. 98, pp. 618–622. ACM (1998)
91. Song, J., Dou, W., Cui, Z., Dai, Q., Wang, W., Wei, J., Zhong, H., Huang, T.: Testing Database Systems via Differential Query Execution. In: International Conference on Software Engineering (ICSE). pp. 2072–2084 (2023). https://doi.org/10.1109/ICSE48619.2023.00175
92. Sørensen, J.J.W.H., Pedersen, M.K., Munch, M., Haikka, P., Jensen, J.H., Planke, T., Andreasen, M.G., Gajdacz, M., Mølmer, K., Lieberoth, A., Sherson, J.F.: Retraction note: Exploring the quantum speed limit with computer games. Nature **584**(7821), 484–484 (Aug 2020). https://doi.org/10.1038/s41586-020-2515-2
93. Steinhöfel, D., Zeller, A.: Input invariants. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 583–594. ACM (2022)
94. Steinhöfel, D., Zeller, A.: Language-based software testing. Commun. ACM **67**(4), 80–84 (2024)

95. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In: Annual Network and Distributed System Security Symposium (NDSS). The Internet Society (2016), http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf
96. Sun, C., Le, V., Su, Z.: Finding and analyzing compiler warning defects. In: International Conference on Software Engineering (ICSE). pp. 203–213. ACM (2016). https://doi.org/10.1145/2884781.2884879
97. Sun, K., Xu, Y.E., Zha, H., Liu, Y., Dong, X.L.: Head-to-tail: How knowledgeable are large language models (LLMs)? A.K.A. will LLMs replace knowledge graphs? In: Conference of the North American Chapter of the Association for Computational Linguistics NAACL-HLT. pp. 311–325. Association for Computational Linguistics (2024)
98. Tao, Q., Wu, W., Zhao, C., Shen, W.: An automatic testing approach for compiler based on metamorphic testing technique. In: Asia Pacific Software Engineering Conference (APSEC). pp. 270–279. IEEE Computer Society (2010). https://doi.org/10.1109/APSEC.2010.39
99. Tao, Y., Chen, Z., Liu, Y., Xuan, J., Xu, Z., Qin, S.: Demystifying "bad" error messages in data science libraries. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering ESEC/SIGSOFT FSE. pp. 818–829. ACM (2021)
100. Tomaszuk, D., Kellogg, G.: RDF 1.2 Turtle. W3C working draft, W3C (Oct 2024), https://www.w3.org/TR/2024/WD-rdf12-turtle-20241031
101. Tomaszuk, D., Kellogg, G.: RDF 1.2 Turtle (outdated). W3C working draft, W3C (Oct 2024), https://www.w3.org/TR/2024/WD-rdf12-turtle-20241024
102. Vecovska, M., Jovanovik, M.: RDFGraphGen: A synthetic RDF graph generator based on SHACL constraints (2024), https://arxiv.org/abs/2407.17941
103. Vrandecic, D.: Ontology evaluation. In: Handbook on Ontologies, pp. 293–313. International Handbooks on Information Systems, Springer (2009)
104. Wang, H., Wang, J., Wang, J., Zhao, M., Zhang, W., Zhang, F., Xie, X., Guo, M.: GraphGAN: Graph representation learning with generative adversarial nets. In: AAAI Conference on Artificial Intelligence. pp. 2508–2515. AAAI Press (2018). https://doi.org/10.1609/AAAI.V32I1.11872
105. Winterer, D., Su, Z.: Validating SMT solvers for correctness and performance via grammar-based enumeration. Proc. ACM Program. Lang. **8**(OOPSLA2), 2378–2401 (2024). https://doi.org/10.1145/3689795
106. Winterer, D., Zhang, C., Su, Z.: Validating SMT solvers via semantic fusion. In: International Conference on Programming Language Design and Implementation PLDI. pp. 718–730. ACM (2020)
107. Xue, B., Zou, L.: Knowledge graph quality management: A comprehensive survey. IEEE Trans. Knowl. Data Eng. **35**(5), 4969–4988 (2023)
108. Yang, R., Zheng, Y., Tang, L., Dou, W., Wang, W., Wei, J.: Randomized differential testing of RDF stores. In: International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 136–140 (2023). https://doi.org/10.1109/ICSE-Companion58688.2023.00041
109. You, J., Ying, R., Ren, X., Hamilton, W.L., Leskovec, J.: GraphRNN: Generating realistic graphs with deep auto-regressive models. In: International Conference on Machine Learning (ICML). Proceedings of Machine Learning Research, vol. 80, pp. 5694–5703. PMLR (2018), http://proceedings.mlr.press/v80/you18a.html

110. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In: Security Symposium. pp. 745–761. USENIX Association (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/yun

111. Zelenov, S.V., Zelenova, S.A., Kossatchev, A.S., Petrenko, A.K.: Test generation for compilers and other formal text processors. Program. Comput. Softw. **29**(2), 104–111 (2003). https://doi.org/10.1023/A:1022904917707

112. 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

113. Zheng, Y., Dou, W., Tang, L., Cui, Z., Song, J., Cheng, Z., Wang, W., Wei, J., Zhong, H., Huang, T.: Differential Optimization Testing of Gremlin-Based Graph Database Systems. In: Conference on Software Testing, Verification and Validation (ICST). pp. 25–36 (2024). https://doi.org/10.1109/ICST60714.2024.00012

114. Zheng, Y., Dou, W., Wang, Y., Qin, Z., Tang, L., Gao, Y., Wang, D., Wang, W., Wei, J.: Finding bugs in Gremlin-based graph database systems via Randomized differential testing. In: International Symposium on Software Testing and Analysis. pp. 302–313. ACM (2022). https://doi.org/10.1145/3533767.3534409

115. Zhong, R., Chen, Y., Hu, H., Zhang, H., Lee, W., Wu, D.: SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In: Computer and Communications Security (CCS). pp. 955–970 (2020). https://doi.org/10.1145/3372297.3417260

116. Zhuang, Z., Li, P., Ma, P., Meng, W., Wang, S.: Testing Graph Database Systems via Graph-Aware Metamorphic Relations. Proc. VLDB Endow. **17**(4), 836–848 (Mar 2024). https://doi.org/10.14778/3636218.3636236