Mutation-Based Testing of Knowledge Graphs

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

[1]University of Oslo, Oslo, Norway.
[2]IT University of Copenhagen, Copenhagen, Denmark.

Contributing authors: tobiasj@posteo.de; einarj@ifi.uio.no;
eduard.kamburjan@itu.dk;

Abstract

With the advent of AI-driven applications, testing faces new challenges when it comes to the integration of software with AI components. We present a novel testing approach to tackle the integration of software with symbolic AI in the form of knowledge graphs (KG). As the KG is expected to change during the run- and lifetime of the software, we must ensure the *robustness* of the system w.r.t. changes in the KG. Starting with a single KG, we mutate its content and test the unchanged software with the original test oracle. To address the specific challenges of KGs, we introduce two additional concepts. First, as generic mutations on single triples are too fine-grained to reliably generate a KG describing a different, consistent KG, we introduce *domain-specific mutation operators* that manipulate subgraphs in a domain-adherent way. Second, we need to specify those parts of the knowledge graph that the software relies on for correctness. We introduce the notion of a *robustness mask* to describe shapes in the graph to which the mutant must conform. We evaluate our approach on two software applications from the robotic and simulation domain that tightly integrate with their respective KG, as well as three OWL reasoners, where we found several previously unknown bugs.

Keywords: Integration Testing, Knowledge Graphs, Mutations

1 Introduction

The wide and rapid adoption of Artificial intelligence (AI) poses challenges for testing the integration of AI components into software. Not only are these AI components black boxes, they are also expected to change and evolve during the run- and lifetime of an application. The software relies on certain structures in the AI component, yet

must be robust with respect to changes in the component. Knowledge graphs (KG) [1], and related technologies such as ontologies, are a form of symbolic AI based on graph data that are notorious for posing difficulties in software integration, due to the number and complexity of reasoners, query engines and other tools that can be used to operate on them. Their integration in software applications has been recognized as the next big challenge in the semantic web field [2].

Challenges

KG components and inputs of systems-under-tests (SUT) pose several challenges, due to the nature of the interactions between software and KGs.

1. KGs are accessed through reasoners and query engines that use logical reasoning based on ontologies and description logics, turning them into black boxes from the perspective of the software. But while the KG is accessed as a black box, the software still relies on some knowledge and structures within the KGs, such as a specific vocabulary and minimal domain knowledge. For example, a robot control system might implicitly expect that the KG contains a node that represents the robot itself.
2. KGs do not change arbitrarily and not in small steps. Triples in a KG are manipulated according to specific patterns [3] in conjunction with other triples.
3. KGs are *logically embedded*. Not every knowledge graph is also valid—many systems require it to be logically consistent with respect to some ontology.

The first challenge is addressed by general approaches to gray- and black-box testing, but the later two challenges prohibit a direct application of generation-based fuzzing.

Approach

Our approach is based on test-case generation with two key components: (i) a set of mutation operators to generate random test cases, and (ii) a robustness mask, which filters the generated test cases to those that are considered valid KGs.

The *mutation operators* are used to generate test cases by applying a sequence of mutation operations to an initial KG. The initial KG is known to be handled correctly by the software and contains general information that is required. We use two kinds of mutation operators. The first kind are *domain-independent* mutation operators, which are generic and can be applied to every input KG. The second kind are *domain-specific* mutation operators, which we introduce to efficiently generate test cases that are more interesting for the specific software (reflecting the specific application domain). These operators can be more complex and involve several changes to the KG, thus addressing the second challenge. An example for such a mutation can be seen in Figure 1 in Section 3.

The different kinds of operators have different purposes in the generation of test cases. The domain-independent operators ensure that every possible input KG can be generated, while the domain-dependent operators guide us towards relevant test cases.

The *robustness masks* restrict the set of test cases to the ones that are meaningful by filtering out test cases that do not satisfy them. A mask is a set of graph shapes expressed in SHACL [4] that must be validated on the KG. In general, the more robust a software is, the fewer graph shapes are in the mask, as there are fewer restrictions on

the KGs that the software handles correctly. The robustness masks provide a way to specify knowledge that is necessary for the software, thus addressing the first challenge. Furthermore, the KG needs to be consistent with the respective ontology in most cases. By using a reasoner to determine consistency, we address the third challenge. A robustness mask must be developed manually, and we describe an iterative process to do so.

To scale testing of KGs and assess the quality of a test suite in a black-box setting, we provide two additional concepts. For one, we provide a KG *minimization tool* that can reduce the size of input KGs drastically when identifying the root cause of a bug. For another, we measure the *input coverage* by considering the amount of terms from the ontology of the KG that are used. Based on this approach, we aim to answer the following research questions in our work:

- **RQ1:** How can one *test the integration* of software with KGs?
- **RQ2:** Which *restrictions of the integration* of the KG and the software can be characterized using mutation operators and robustness masks?
- **RQ3:** What are the consequences of using *domain-specific mutation operators* compared to domain-independent mutation operators?
- **RQ4:** To what extent can mutation operators guarantee coverage of input features?
- **RQ5:** How much can input KGs be reduced in size to reveal root causes?
- **RQ6:** Can an *iterative process* be used to develop a robustness mask that characterizes the interaction between the software and the KG?
- **RQ7:** To what extent can mutation operators be extracted from reference KGs?

We evaluate our approach on three case studies. First, the SUAVE system for autonomous underwater robots [5] is a robotic system that uses a non-query based approach to interact with a knowledge graph to navigate its environment. We are able to precisely capture possible changes in the environment in domain-specific mutation operators and the assumptions on the knowledge graph in graph shapes. Second, a simulator for geological processes [6], where we are able to capture assumptions on the ontology that specifies the interface between software and knowledge graph. Third, three widely used reasoners (HermiT [7], Pellet [8], ELK [9]) for the EL profile of OWL, where we identify six previously unknown bugs in the reasoners.

Contributions

Our main contribution is a new approach to test software with KG-components or KG inputs, based on mutation testing. Our evaluation shows that it can be used for integration testing [10] of complex, domain-specific software, as well as general-purpose tools that have KGs as inputs. This article extends an ISSRE'24 conference publication [11], where the robustness mask and domain-specific mutation operators were introduced and evaluated on the SUAVE and the geological simulators. This journal extension provides the following additional contributions.

Input Coverage One way to assess the quality of the generated input in a black box testing setting, is to provide a measure for input coverage. So far, no such measure was inverstigated and in this work we consider *input feature coverage*, which measures how many features of the underlying knowledge graphs the test suite covers in terms of predefined terms. We address feature coverage in the new research question **RQ4**

and discuss how to use our method to maximize this coverage (see Section 5.3). We extended our evaluation by introducing a measurement for input feature coverage and we added two experiments to compute the measurement for different inputs (see Section 6.3).

Automated Test-Case Reduction. The knowledge graphs that trigger bugs or other undesired program behavior can be large and difficult to interpret. Thus, detecting the root cause so far required to manually analyze the mutated knowledge graph. In this work, we introduce a new research question **RQ5** that addresses this. We propose a new component for test-case reduction in our testing architecture (see Section 4.2.2) and evaluate the effectiveness of the proposed reduction technique (see Section 6.3).

Automated Extraction of Mutation Operators. Extracting the domain-specific mutation operators has been the most time-consuming manual process in prior work, as it requires both domain expertise and the ability to interpret structures in the knowledge graph. In this work, we introduce an approach to automatically extract such operators. We introduce the new research question **RQ7**, which addresses the automated generation of mutation operators. We propose an algorithm to extract mutation operators from existing data based on the analysis of common graph patterns [12] and evaluate the quality of the obtained operators.

Test Profiles. Lastly, we introduce a *test profile* to describe how the mutated KGs should be generated and which structures they should have (see Section 4.1, paragraph *test profile*). A test profile contains both the domain-specific operators and the robustness mask, but also allows to configure further settings.

In addition evaluating these new contributions on the previous case studies, we additionally apply our methodology to test widely used OWL reasoners (see Section 6.2), where we found and documented several, previously unknown bugs (see Section 6.3), and confirmed further bugs from other testing campaigns [13]. How to automatically test logical reasoners and solvers is a open research question and has far only address by this work and a prior study by the authors [13], which we compare to it the discussion.

2 Background

Our work targets a research gap that we identified by analyzing the state-of-the-art of testing software with KG. Before discussing these approaches, we introduce some preliminaries.

2.1 Preliminaries

Knowledge Graphs and Ontologies

Knowledge graphs [1] are a technique for knowledge representation that connects graph *data* with ontological *axioms* (the so-called ontology) [14]. Due to the ontological axioms, querying a knowledge graph can invoke a deductive reasoner to deduce new facts about the nodes in the graph data. While powerful, this reasoning makes

modularization and interfacing hard, as one does not a priori know which parts of the ontology are relevant to answer a query.

While our implementation is based on the standard W3C stack of RDF[1] (to represent the graph), OWL[2] (for the ontology), SPARQL[3] (for the queries) and SHACL[4] (for the graph shapes), we use the more concise Description Logic (DL) syntax [15] here. DL is the formal underpinning of OWL-based knowledge graphs, and distinguishes data (denoted ABox) from ontology (denoted TBox) more clearly than RDF, a separation that will come in handy in later sections.

Testing

Testing is a quality assurance technique in software engineering, and can be used at different levels to target either single units of a system (unit testing), the integration of several interacting units (integration testing), or the whole system (system testing) [16]. In this work, we use test cases that consist of a *test input* and a *test oracle*. We consider two types of testing: (i) *integration testing* [10], which determines whether the combination of at least two components fails or passes the test and (ii) *system testing* [17], which determines whether a complete system fails or passes the test.

The quality of a set of test cases can be determined based on different metrics, which fall in one of two categories: either one measures the coverage of the source code of the SUT or one measures the coverage of the set of all possible inputs. The first category measures how well the execution of the SUT on the test cases covers the program structure of the SUT. Examples of such metrics are *function coverage*, *statement coverage* or *path coverage* [18, 19]. The latter category measures how well the test cases cover the space of all possible inputs, i.e., it only depends on the inputs that are allowed for the SUT but not on the SUT itself. In particular, it is of interest to cover with the test cases as many of the different features that can occur in the inputs and as many combinations of those features. It has been shown that a high coverage of the input features and their combinations is beneficial for finding bugs [20, 21].

To create test inputs, one can either hand-curate new test input manually, or generate them automatically, a process called *fuzzing* [22]. Fuzzing can be either mutation-based or generation-based. *Mutation-based fuzzing* creates new test cases based on existing ones by changing parts of them, while *generation-based fuzzing* generates random new inputs that satisfy some constraints. Both forms of fuzzing are suitable to test robustness, the "degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [23], but in highly constrained situations, mutation-based fuzzing can be considered better suited to avoid generating too many invalid inputs [22].

Designing oracles is a challenging task [24], and if generating the expected output is not possible, one can use indirect techniques, such as *metamorphic testing* [25], which relates the outputs of different runs to each other, for example the output of an original test case to a generated one. Another indirect technique is *differential testing* [26], which relies on having at least two different systems that realize the same

[1]Resource Description Framework, www.w3c.org/RDF/, accessed 2025-12-17
[2]Web Ontology Language, www.w3c.org/OWL/, accessed 2025-12-17
[3]SPARQL Query Language for RDF, www.w3.org/2001/sw/wiki/SPARQL, accessed 2025-12-17
[4]Shapes Constraint Language, www.w3.org/2001/sw/wiki/SHACL, accessed 2025-12-17

functionality. After executing both on the same test case, one compares the outputs of both systems and if the outputs are not identical, the oracle is negative as at least one system must have calculated an incorrect result.

2.2 State-of-the-art

Test-driven-development can be applied to ontologies, where the test cases are axioms that must be entailed even after modification and specify certain competency questions, which in turn act as requirements [27, 28]. This procedure is fully independent from the software components eventually using the ontology. Similarly, mutations of ontologies, mostly focusing on the TBox axioms [29, 30], have been proposed to check how robust the ontology is to changes.

Research on testing software working on ontologies and KGs, has focused on systems that work on generic graph data, mainly graph database management systems [31–38]. Most approaches use purely random KGs as input, while only a few leverage a mutation-based approach [32, 37]. None of the methods considers filtering the generated KGs to remove redundant test cases, before using them for testing. As test oracles, differential testing [33, 35, 38] and metamorphic testing [31, 32, 36] are the most common. We use differential testing for one of our testing campaigns in the evaluation (namely, the Reasoners). In this line of work, queries have been used as the input to detect both logical and generic bugs. However, the software and KGs are loosely coupled. Only two approaches considered RDF-KGs [13, 36]. Metamorphic testing of Datalog engines, while not directly addressing KGs, similarly focuses on generating new instances for software components that work on generic inputs [39].

There are several approaches to generate random KGs, without considering using them for testing. The approaches can be classified in two categories: (i) data-driven approaches and (ii) schema-driven approaches. *Data-driven approaches* generate KGs based on existing data about the SUT, such as tabular database data [40], execution traces of the software [41] or the structure of existing reference graphs [42–46]. Our method also relies on having a reference KG, which we use as the seed for the mutations, but we only need one such KG. *Schema-driven approaches* rely on schema information for which a conforming KG is generated [47–49]. We follow the schema-driven approach, using only KGs that conform to a provided schema for testing. Similarly to the RDFGraphGen tool [49], we use SHACL to express the schema but additionally take OWL axioms encoded in the KG into account. In contrast to all the discussed methods, we propose a method where the generated KGs are specific to the SUT.

There are three approaches to generate random KGs that not only generate random data but also random schema information [13, 50, 51]. However, these approaches can only generate a limited set of schema relations mostly focusing on simple class relationships. Our approach can generate KGs that contain any kind of schema information that can be expressed with RDF.

Generation-based fuzzing, e.g., grammar-based fuzzing [52], can generate random *syntactically correct* test cases, as one can utilize the formal grammars underlying the KGs. However, these test cases are not necessarily *semantically correct*, i.e., they are not consistent with the respective ontology (see third challenge) and do not respect the assumed structures within the KGs (second challenge). More complex constraints can

be imposed on grammars by fuzzers [53], but these techniques are still not expressive enough to express arbitrary graph constraints, and are so far only useful for graphs without constraints [13].

Existing mutation-testing approaches for the underlying ontology [29, 30, 54] provide an alternative way to generate random KGs. These approaches use generic mutations, which contain operations like deleting nodes from the KG or swapping two classes in the class hierarchy of the underlying ontology. However, such approaches are limited to single steps of changes and do not ensure that the KG is consistent with the ontology and do not preserve assumed structures, again failing to address the second and third challenge.

Lemieux and Sen use a *branch mask* to control where an input byte sequence may be mutated to increase branch coverage of the test suite [55]. Our approach similarly masks part of the KG to prohibit its mutation, but works on a more structural level than mere bytes.

Research Gap

The above discussion shows that, there is currently no approach for integration testing of systems in which software and KG components interact. In particular, there is no notion of an interface between these components, as competency questions are posed by subject-matter experts during the development of the ontology and do not consider software. Consequently, no way to variate the KG in a meaningful way to generate further test scenarios is known. Additionally, mutations of ontologies so far only consider TBox axioms and completely ignore the ABox instances that must be consistent with them. Thus, it is not possible to assess the robustness of the SUTs.

In the next section, we illustrate by a concrete example how this gap manifests itself for tightly coupled systems – software components that rely on the specifics of the used KGs and interact with them in a fixed vocabulary.

3 Motivating Example

To illustrate tightly coupled and KG-based software applications, their challenges and to give an overview over our approach, consider the software for an autonomous underwater vehicle (AUV), which performs infrastructure inspection in an unknown environment [56, 57]. This is a simplified system based on the deployed SUAVE system, which we will use in our evaluation in Section 6. The KG serves two purposes: First, the KG encodes the perceived environment and contextualizes this environment using domain knowledge encoded in an ontology. Second, the KG encodes the mission and the possible actions of the AUV [58–61].

Infrastructure Inspection

For pipeline inspection, a scenario is depicted in Figure 1. In Figure 1a, we provide an algorithm that orchestrates the inspection of (all) the pipe segments. The algorithm relies on knowledge about the environment, which the algorithm accesses using queries (marked with **query**(...)) to retrieve sets of elements for further processing. The algorithm starts by retrieving the current position, inspects the pipe segment

that the AUV is at and finds an adjacent segment of pipe that has not been inspected yet. Afterwards, the AUV moves to the new pipe segment. This is repeated until no uninspected adjacent pipe segment can be found.

For the representation of the knowledge, the KG in Figure 1c shows how we can encode an environment as a graph. The picture shows an AUV in an underwater environment with two pipe segments. Next to that, we show how this scenario can be encoded as a knowledge graph. All relevant entities in the scenario, i.e. the AUV and the pipe segments, are expressed as nodes that are connected by appropriate relations to explain their location w.r.t. each other. Furthermore, we might add information about the classification of the individuals, e.g. the pipe segments are all of the class `:Pipe`. The SUT successfully inspects all pipe segments in this scenario (first, it inspects `:p1` and then `:p2`).

Challenges

Testing the robustness of the integration of the software component implementing the algorithm with the KG involves the challenges mentioned in Section 1.

(1) Although the software treats the KG as a black-box, it implicitly has some assumptions about its structure. Some of these assumptions are rather obvious, e.g. that the AUV is in the beginning at the position of exactly one pipe segment, others are more subtle. One such assumption is that the inspected pipe structure is not branching. The illustration in Figure 1d contains a scenario, i.e. a knowledge graph, where this assumption is violated. The algorithm does not behave as desired when it uses this KG as the input: After inspecting the first segment, either the second or the third segment will be visited. However, from both of them, there is no adjacent uninspected segment as the first segment is already inspected. Thus, the SUT terminates without inspecting all pipe segments.

(2) The KG in Figure 1d, which reveals an assumption of the software, differs in several ways from the original KG in Figure 1c. Multiple changes need to be made to the original KG to obtain the new KG: a new individual of the type `:Pipe` needs to be added and the `:nextTo` relations between the new segment and the first one need to be added. Having a test case where only one of the changes is considered, e.g. only adding a new individual with a `:nextTo` relation to the first segment, might result in a KG that does not represent a meaningful scenario, e.g. because only individuals of type `:Pipe` can be in a `:nextTo` relationship. Thus, the changes between relevant test cases are non-continuous.

(3) The graph structure has to be consistent with the respective ontology. This might require that only the node `:auv` is in an `:isAt` relationship with some other node and that the node `:auv` is not of type `:Pipe`. Our testing framework must ensure that we only generate consistent KGs.

Overview

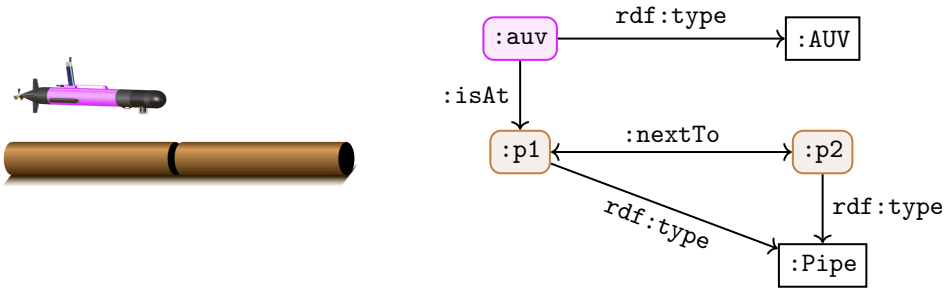
To test the robustness of the integration, we need a generator for KGs that are within the domain of the implemented algorithm. To construct those, we start with the KG in Figure 1c and then use mutation operators that are specifically designed for the underwater domain.

```
p := query(":isAt(:auv, ?p)")
inspect(p)
S := query(":nextTo(p, ?s)")
while S ≠ ∅ do
    p := S.pop()
    if ¬inspected(p) then
        moveTo(p)
        inspect(p)
        S := query(":nextTo(p, ?s)")
    end if
end while
```

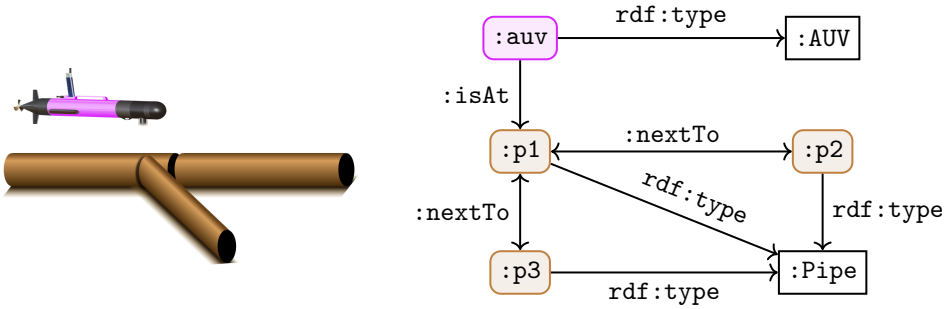
(a) Pseudo code of an algorithm for pipeline inspection, which accesses the KG via queries marked with **query**.

```
AuvAtPipeline
  a sh:NodeShape ;
  sh:targetNode :auv ;
  sh:property [
    sh:path :isAt ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:class :Pipe ;
  ].
```

(b) Robustness mask that describes a restriction on the shape of the KG, namely how `:auv` needs to be connected to other nodes.



(c) Scenario with corresponding KG representation before mutating.



(d) Scenario with corresponding KG representation after mutating.

Fig. 1: Components for a simple pipeline-inspection algorithm: Pseudo code, graph representations for original / mutated scenarios and a robustness mask.

We apply *domain-specific mutation operators* to the original KG (Figure 1c) to create test cases. These mutations should be able to generate test cases that reveal new information about the behavior of the software, like the one in Figure 1d. So, we can define an operator that generalizes the change from the KG in Figure 1c to the KG in Figure 1d. The result is a mutation operator for adding a new pipe segment, which is applicable to every existing individual of type `:Pipe`. In general, a domain-specific mutation operator works on some subset of the KG. This operator includes several changes that are all necessary to reflect the addition of a new pipe segment and thus solves the challenge of the non-continuous behavior of changes of the KGs. The operator is only relevant for the specific domain of pipe infrastructure inspection.

We use *robustness masks* to describe declaratively the scenarios in which the software works as intended. It is thus a measure to describe the assumptions that the software has about the KG. The less robust the software is, the more restrictive is its mask. The mask contains shapes that must be enforced by the KG together with the underlying ontology. An example for a mask in our scenario is depicted in Figure 1b. It contains a restriction on how the node `:auv` is connected to other nodes. In particular, that is exactly in one `:isAt` relation with another node and this node is of type `:Pipe`. This restriction needs to be satisfied, as the algorithm fails otherwise because the initial position of the AUV can not be extracted from the KG.

Before it is used for testing, a generated KG is checked against the mask of the software, as we are only interested in KGs that reveal new information about the software and not in test cases for which we already know that the software does not behave as intended.

Additionally, KGs are checked for consistency with their respective ontology. Using domain-specific mutation operators is a way to ensure that most of the generated KGs are consistent, while generic mutations will often generate inconsistent KGs. For example, a generic mutation might add new `:nextTo` relations between nodes, which can lead to an inconsistency if we consider the KG on the top and an ontology that requires that only nodes of type `:Pipe` can be in a `:nextTo` relation.

Generating test cases where the software fails, such as the one in Figure 1d, reveals that the software is less robust than assumed. It provides evidence that the robustness mask does not describe the valid inputs correctly. In such a situation, one must update the mask. In our example, we need to add a shape that the pipe structure is not branching to the robustness mask. We discuss the creation of such a mask in detail in Section 5.5.

4 Testing Architecture

We propose to use the *testing architecture*, which is shown in Figure 3 to test the integration of software with KGs. We first introduce the relevant structures and the different components before explaining the workflow of using our architecture.

4.1 Structures

Our testing architecture uses different structures to represent KGs, software, mutation operators and graph shapes.

Knowledge Graphs

We follow the usual view on RDF graphs: Each element (nodes or types of relations) have their own unique identifier, also known as an IRI, which can be an arbitrary string. The graph structure is a set of triples of these identifiers. Each triple represents a directed edge in the graph, with a start node, a relation that connects the nodes and a target node (where the edge ends). A *subgraph of a KG* is a subset of the set of triples that make up the KG.

Software

The software that we consider takes a KG as an input on which their behavior depends. We can view the (observable) behavior as the output of the software. We use this output to judge whether the software works correctly. So in general, a software takes a KG and produces an output. Hence, we can view software as functions that map a KG to one of the possible outputs.

Example 1. *Consider the pipe inspection example from Section 3. We can view the number of inspected pipe segments as the output of the software; i.e., the possible outcomes are the natural numbers. The software works correctly if the output equals the number of pipe segments in the KG.*

Mutation Operators

A mutation operator for KGs selects a subgraph and replaces it by a different subgraph. To make the operators applicable to different KGs, we use *graph patterns*, which include variables in addition to the unique identifiers from the KG. This allows a mutation operator to be specified in an abstract manner. A pattern is a set of triples; i.e., it is a graph itself. To describe actual subgraphs of a specific KG, we instantiate the patterns by mapping the free variables to elements of the KG. We call such subgraphs *instantiations* of the mutation operator. Because instantiated graph patterns do not contain any variables, they are essentially KGs.

Example 2. *We consider an example related to the domain in Section 3. We use the graph pattern* $\{(x, \text{type}, \text{Pipe}), (y, \text{type}, \text{Pipe})\}$*, which describes two individuals of type* Pipe*. Instantiating the pattern with the valuation that maps* x *to* p1 *and* y *to* p2 *yields the new pattern* $\{(\text{p1}, \text{type}, \text{Pipe}), (\text{p2}, \text{type}, \text{Pipe})\}$*, which does not contain any variables.*

A mutation operator is a pair of two graph patterns (S, R). The pattern S describes which subgraphs should be selected to apply the operator, i.e. where to mutate the KG. The pattern R describes how the selected subgraph should be replaced, i.e. how to mutate the KG. All possible instantiations of the first pattern that are subgraphs of the KG are possible places to apply the operator. The second pattern describes the new graph structure that should replace the found subgraph. Note, that the new pattern can contain more variables than the pattern with which we select the subgraph to apply the mutation. Such additional variables can match either: any existing node in the KG or new individuals whose identifier has not been used so far in the KG.

```
SimplePipeShape
  a sh:NodeShape ; sh:targetClass Pipe ;
  sh:property [
    sh:path nextTo ; sh:minCount 1 ;
    sh:class Pipe ; ].
```

Fig. 2: Example of a SHACL shape for the pipeline example.

Example 3. *We consider again the example from the pipeline domain. We define a mutation operator* (S, R) *that connects two pipe sections with the patterns*

$$\mathrm{S} = \{(x, \mathrm{type}, \mathrm{Pipe}), (y, \mathrm{type}, \mathrm{Pipe})\} \qquad \textit{and}$$
$$\mathrm{R} = \{(x, \mathrm{type}, \mathrm{Pipe}), (y, \mathrm{type}, \mathrm{Pipe}), (x, \mathrm{nextTo}, y)\}.$$

Note, that the pattern R contains all triples from S; otherwise, they would be deleted when the operator is applied. We consider the following KG as before, containing two pipe segments: $\mathcal{G} = \{(\mathrm{p1}, \mathrm{type}, \mathrm{Pipe}), (\mathrm{p2}, \mathrm{type}, \mathrm{Pipe})\}$*. The operator describes four mutants for this graph, resulting from the four possible ways to map the two variables to the two pipe segments. The mutants are the following KGs that all contain the original KG and one additional relation:* $\mathcal{G} \cup \{(\mathrm{p1}, \mathrm{nextTo}, \mathrm{p1})\}$*,* $\mathcal{G} \cup \{(\mathrm{p1}, \mathrm{nextTo}, \mathrm{p2})\}$*,* $\mathcal{G} \cup \{(\mathrm{p2}, \mathrm{nextTo}, \mathrm{p1})\}$ *and* $\mathcal{G} \cup \{(\mathrm{p2}, \mathrm{nextTo}, \mathrm{p2})\}$*.*

The described mutation operators are defined quite general. From a practical point of view, one can further classify the operators based on the subgraph that is affected by the mutation: the selected subgraph can be only concerned with the nominal data, i.e., the ABox, or only with universals, i.e., the TBox, or with both. In our example, adding a pipeline segment corresponds to selecting nominal data, while changing a subclass relation corresponds to modifying universals. Creating a new subclass of pipeline segments and changing some specific segments to be members of the new class falls into the last category. Thus, all three possibilities are potentially useful.

Graph Shapes

We use SHACL to define shapes of KGs[4]. A SHACL shape defines the structure of a subgraph and where they have to be used in the KG. If a KG contains the specified structures at the required places, it *conforms* to the shape. A SHACL shape is expressed as a KG itself.

Example 4. *We consider again the graph and mutation operator from Example 3. We assume that we applied the mutation operator once to generate the mutant* $\mathcal{M} = \{(\mathrm{p1}, \mathrm{type}, \mathrm{Pipe}), (\mathrm{p2}, \mathrm{type}, \mathrm{Pipe}), (\mathrm{p1}, \mathrm{nextTo}, \mathrm{p2})\}$*. The SHACL shape in Figure 2 describes the requirement that every pipe segment needs to be connected to at least one other node, which has to be a pipe segment. The mutant does not conform to the shape as* p2 *is not in a* nextTo*-relation. However, by applying the mutation operator twice to the original KG, we can generate the mutant* $\mathcal{M}' = \mathcal{M} \cup \{(\mathrm{p2}, \mathrm{nextTo}, \mathrm{p1})\}$*, which conforms to the shape.*

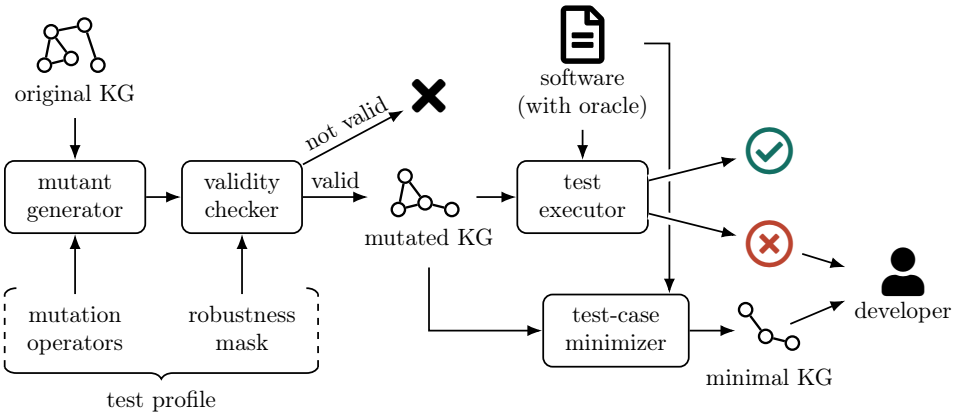


Fig. 3: Testing architecture.

Test Profile

The *test profile* contains information about how to do the mutation and how to check, if a KG should be used for testing. The test profile contains two elements: (i) a list of candidate mutation operators and (ii) a *robustness mask*, which is a set of SHACL shapes. The first element describes, how a KG should be mutated, i.e., where the mutations can be applied and how the graph is modified when they are applied. The second element, the robustness mask, restricts the application of the mutation operators: it defines shapes that the resulting mutant KG needs to conform to, i.e., it defines structures that need to be preserved by the mutation operators. Together, the elements define precisely, what kind of mutants are allowed to be generated when the testing is performed.

Example 5. *To construct a test profile, we combine the mutation operator defined in Example 3 and use the graph shape from Example 4 as the robustness mask. While the mutation operator adds connections between nodes that are of type* Pipe*, the robustness mask ensures that each pipe is connected to at least one other pipe. Together, they describe changing a pipe network by adding new connections while ensuring that the resulting network consists of a single connected component.*

4.2 Components

Our testing architecture is built out of several components. We introduce them one by one, explain their behavior and how they connect to each other.

4.2.1 Required Components

We first discuss the components that are required in the architecture. These are the mutant generator, the validity checker and the test executor.

Mutant Generator

The *mutant generator* is the central component of our architecture. Its location in the architecture is depicted in Figure 3. It takes the KG that is going to be mutated, called the *seed*, as an input. To perform mutations, it further needs a (finite) set of mutation operators and the desired number n of mutations applied to the initial KG to obtain the mutated KG, called the *mutant*. The mutant generator starts by randomly selecting a mutation operator. The operator might have several or no place to be applied to the KG. If the operator is applicable, i.e. there is at least one mutant of the KG described by the operator, the mutant generator selects randomly one of the mutants described by the operator. Otherwise, a different operator is chosen. Afterwards, the mutant generator starts with the mutant and applies a new operator to obtain the next iteration of the mutant. This procedure is repeated until n operators have been applied to the KG. The higher the number n, the more likely it is that the test run will fail, as the mutant is more different from the original KG. This reduces the number of times the test executor needs to be run, which is desired, as running the software can be time consuming. The resulting mutant is the output of the mutant generator.

Validity Checker

The *validity checker* is a component that takes the mutant generated by the mutant generator and checks, if the mutant can be used for testing (see Figure 3). To do so, the component takes a *robustness mask*, which is a set of SHACL shapes, as an input. It tests, whether the generated mutant conforms to all the SHACL shapes in the mask.

Additionally to checking if the mutant conforms to the robustness mask, the mutant can be checked to be consistent w.r.t. the ontology it contains. An inconsistent KG entails all knowledge and is therefore useless for many applications. In such a case, we only classify a KG as valid if it is consistent.

The result of the validity check determines if the mutant is considered for testing or not. Only if the result is positive, the mutant is handed forward to the next component.

Test Executor

The last required component in our architecture (see Figure 3) is the *test executor*. It takes the generated mutant and uses it as input for the provided software component. The output of the software execution is then compared with the oracle for the software with the mutant. Depending on whether the two outputs are the same or different, the mutant is classified as passing or failing. This result is the overall result of the architecture to be considered by the developers.

4.2.2 Optional Component: Test-Case Minimizer

Minimizing test cases is an important aspect of software testing to identify the cause of an undesired behavior. We propose a component that can be used optionally to do this minimization of KGs. Especially when working with large KGs, not all triples in the graph might be relevant to reproduce a negative oracle outcome. To make root cause analyses simpler, it is desirable to identify which triples really cause the negative oracle. The *test-case minimizer* removes all triples from a KG that are not required. It

Algorithm 1 Algorithm to minimize KG $\mathcal{G}$ while preserving outcome of running software $\mathcal{P}$.

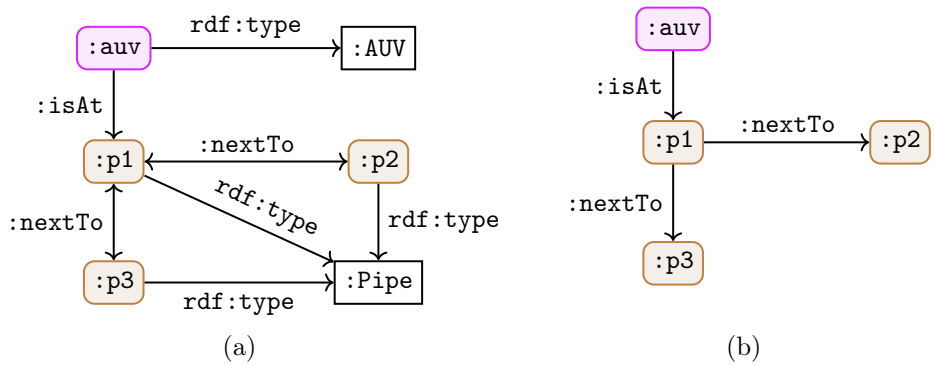
procedure MINIMIZEKG($\mathcal{G}$, $\mathcal{P}$)
  $\mathcal{O} \leftarrow \mathcal{P}(\mathcal{G})$
  return MinimizeKG($\emptyset$, $\mathcal{G}$, $\mathcal{P}$, $\mathcal{O}$)
end procedure

procedure MINIMIZEKG(S, $\mathcal{G}$, $\mathcal{P}$, $\mathcal{O}$)
  if $|\mathcal{G}| = 1$ **then**
    return $\mathcal{G}$
  end if
  $S_1, S_2 \leftarrow \text{split}(\mathcal{G})$
  if $\mathcal{P}\ (S \cup S_1) = o$ **then**
    return MinimizeKG$(S, S_1, \mathcal{P}, \mathcal{O})$
  end if
  if $\mathcal{P}\ (S \cup S_2) = \mathcal{O}$ **then**
    return MinimizeKG$(S, S_2, \mathcal{P}, \mathcal{O})$
  end if
  $S_1^{\min} \leftarrow$ MinimizeKG$(S \cup S_2, S_1, \mathcal{P}, \mathcal{O})$
  $S_2^{\min} \leftarrow$ MinimizeKG$(S \cup S_1^{\min}, S_2, \mathcal{P}, \mathcal{O})$
  return $S_1^{\min} \cup S_2^{\min}$
end procedure

is based on a well-established algorithm that is designed to compute a justification for an inconsistent set of axioms [62, 63], i.e., an algorithm that minimizes an inconsistent set of logical axioms such that the result is still inconsistent. We modify this algorithm slightly: instead of ensuring inconsistency, we ensure that the outcome of the software remains the same. However, the core structure of the algorithm remains the same.

Algorithm

The algorithm is shown in Algorithm 1 and the procedure MinimizeKG($\mathcal{G}$, $\mathcal{P}$) is called to minimize a KG $\mathcal{G}$ w.r.t. to a software $\mathcal{P}$. First, the procedure computes the outcome of the software on the original KG. This outcome, together with the software itself is provided to the second procedure. This procedure works recursively by partitioning the set of all triples in the KG in half. The partitioning is random but ensures that S_1 and S_2 are of equal size.After splitting the set of triples, the algorithm checks if the outcome of running the software remains the same when only considering one of the two subsets. If yes, the algorithm discards the other subset and continues by minimizing the subset that leads to the same outcome. In the case that the outcome of the software of both subsets differs from the original outcome, the minimal set has to contain triples from both subsets. In this case, the algorithm first minimizes the first subset (while keeping all triples from the second subset), which leads to a minimized version of the first subset. Secondly, the algorithm minimizes the second subset (while keeping all triples from the minimized version of the first subset). At the end, the



(a) (b)

Fig. 4: Minimization of the bug triggering input from the example in Section 3. (a) is the original input KG and (b) is the minimized KG.

minimized versions of both subsets are combined into the final, minimal set of triples by taking the union of the two sets. This set of triples is again a KG.

The algorithm is optimal in the sense that no triple can be removed from the result without leading to a different outcome of the test run, i.e., the calculated result is minimal. However, it is not guaranteed that there is no KG with fewer triples that also satisfies this property, i.e., the algorithm finds a local minimum but not necessarily the global minimum. The computed KG depends on how the KGs are split into two sets of triples. Because the splitting is random, running the algorithm multiple times can lead to different results, with different numbers of triples.

To compute the minimal set of triples, the test-case minimizer runs the software $\mathcal{P}$ with different subsets of the original KG and thus it needs to have access to the software that is tested. The runtime complexity of the algorithm depends on the complexity of running the software $\mathcal{P}$ on a KG $\mathcal{G}$. In particular, it depends on the runtime complexity of $\mathcal{P}$ with respect to the size of $\mathcal{G}$, i.e., the number of triples in $\mathcal{G}$. An analysis using the master theorem for divide-and-conquer algorithms [64] can be used to compute the complexity of the minimization algorithm. This runtime complexity differs for different software $\mathcal{P}$. We name two examples of common complexity classes. If the runtime of $\mathcal{P}$ is linear in the size of the KG, the overall complexity of the minimization algorithm is $\Theta(|\mathcal{G}|\log(|\mathcal{G}|))$. If the runtime complexity of $\mathcal{P}$ is within $\Omega(|\mathcal{G}|^n)$ where $n > 1$, i.e., it is at least polynomial with an exponent larger than 1, the overall complexity of the minimization algorithm is the same as running $\mathcal{P}$ on the initial KG $\mathcal{G}$. This shows that the runtime complexity of the minimization algorithm is in many cases either within $O(|\mathcal{G}|\log(|\mathcal{G}|))$ or bounded by the runtime complexity of $\mathcal{P}$ on the initial test input. Therefore, it is feasible to run the minimization algorithm on the same setup as the testing campaigns for $\mathcal{P}$. We discuss in Section 6.3 how effective this minimization component is in removing unnecessary triples in practice.

Example

We consider again the example from Section 3. Figure 4 shows the effect of using the test-case minimizer. The KG on the left was obtained using mutation operators and

triggers a bug in the tested algorithm. As consequence, the robot does not inspect all pipe segments. However, not all triples and nodes of the KG are necessary to trigger the bug. It is enough to have some node `:auv` that is at some segment with two outgoing `:nextTo` relations. In fact, all other triples are irrelevant. Thus, the minimization component produces the KG on the right in Figure 4. The minimized KG contains only three triples, while the original KG contains nine triples, i.e., six triples can be removed, which corresponds to 66% of the test case.

4.3 Workflow

We require that the user of the architecture has access to the software component to use our architecture to analyze the SUT. Furthermore, the user should provide a KG for which the software is known to work correctly. This KG is provided as the seed to the mutant generator and is the foundation for all generated mutants. Furthermore, the user needs to specify a test profile, which involves listing the desired mutation operators, and providing the mask to use.

The mutation operators can be of two kinds: *domain-independent* or *domain-specific*. Formally, they differ by the vocabulary that is used. Domain-independent operators only use identifiers that are commonly used, e.g. included in the OWL standard. Some of these operators can be obtained from descriptions in existing work [29, 30]. On the other hand, the domain-dependent operators use identifiers that are only defined specifically for the domain of the software. Such operators need to be defined by the user and should reflect changes in the graph that are considered interesting, i.e. for which the user expects to gain insights into the interaction of the software component and the KG, based on the outcome of the test. Domain-dependent operators often involve multiple changes to the KG while domain-independent operators are usually more simple as they need to be general enough to be applied in many domains. If the user wishes to ensure that the wide variety of possible KGs is explored, the domain-independent operators for adding and deleting nodes and relations should be part of the set of operators.

The intended number of mutations is also considered by the mutant generator. This number depends on the software and how different the mutants should be from the initial KG.

If some restrictions for the KG are already known, the user provides them in the test profile via the robustness mask that characterizes them. The SHACL shapes in the mask describe information about the SUT that is already known, e.g. that some triples need to be part of the KG, or if only some test cases are considered relevant for testing, e.g. for the pipeline inspection case one might be only interested in cases where there is at least one robot. If an empty mask is provided, e.g. because no restrictions are known, all mutants are used for testing. More details about how the robustness mask can be obtained are described in Section 5.5.

The last thing that the user needs to provide is a test oracle for the mutated KG. Depending on the software, this can be rather simple or a difficult problem but providing an oracle is a common requirement. If the test run does not comply with the oracle, the test run indicates that the provided robustness mask does not capture the restrictions of the SUT correctly. Hence, a mutant where the output of the software

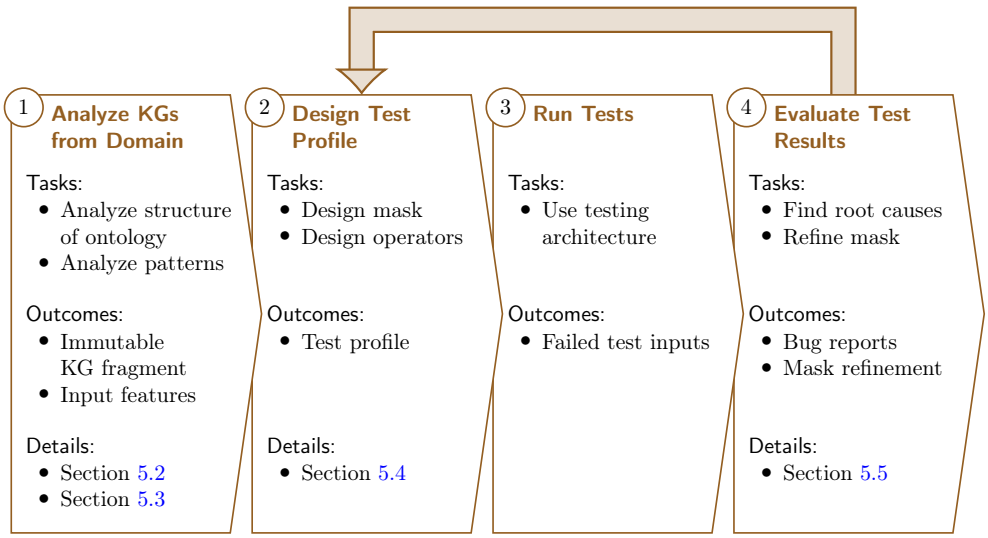


Fig. 5: The proposed use of the testing architecture.

does not comply with the oracle needs to be interpreted by the developer and is the source of new insights about the interaction, that can be added to the robustness mask. The test-case minimizer might be used to remove all unnecessary triples from the KG and to provide a minimal version of the KG that triggered the negative oracle to the developer.

5 Testing Methodology

To use our testing architecture most effectively, we propose to use it as describe in this section. After providing an overview how to use the testing architecture, we present details on different aspects of this process. We discuss to which parts of the KG the mutation operators should be applied, how to select mutation operators to cover input features, how to extract mutation operators from reference KGs, how to obtain a robustness mask and how to interpret the generated mask.

5.1 Usage of Testing Architecture

The overall process for using the testing architecture is shown in Figure 5. The central step is to design the test profile and potentially iterate over it.

① The first step is to analyze reference KGs from the domain of interest. This step requires domain expertise. An analysis of the structure of the ontology in the reference KGs can lead to the identification of a common fragment that all KGs share and that should be immutable (see Section 5.2). Similarly, an analysis of common patterns in the KGs can lead to the identification of the input features of the KGs (see Section 5.3). These insights, together with a general understanding of the domain, form the basis of the next step.

② The second step is to design the test profile. Both components of the test profile, i.e., the mutation operators and the robustness mask, are influenced by the information about immutable KG fragments: Masks ensure that the immutable parts of the seed KG are present in mutant KGs and the mutation operators should be designed to not modify the immutable part.

Mutation operators for the test profile can be obtained in different ways. Most importantly, the mutation operators should be designed such that the domain-specific patterns are preserved, which ensures that the mutant KG is within the domain of interest. Additionally, the input features identified in the first step can be used as guidance when defining the mutation operators (see Section 5.3).

Overall, domain expertise plays a bigger role in the design of the mutation operators than for the robustness mask. The reason is that one can define an initial mask, which allows to obtain test results and refine the test profile later. Domain-specific mutation operators are necessary to generate relevant test inputs in the first place. Without mutation operators, no targeted testing is possible. While the masks increase the efficiency of the testing by filtering test inputs, only the mutation operators can lead to a coverage of all inputs of interest. Because our test methodology crucially depends on the mutation operators and the design process is a priori labor intensive, we propose an automated approach to extract mutation operators from reference KGs (see Section 5.4).

③ The third step is to run the tests using the test architecture described in Section 4. The reference KGs collected for the first step are used as seeds for mutations. As the test profile was already defined in the second step, this third step is fully automatic. The result is a collection of test inputs for which the verdict is negative.

④ The last step is to evaluate the results of the test runs. The test-case minimizer is used in this step to simplify large KGs. This makes it easier to identify the root causes for negative verdicts.

Additionally, this fourth step can involve a refinement of the test profile. A refinement of the robustness mask can later be used to avoid generating mutants that trigger already known bugs. We detail the process of developing the mask using an iterative process in Section 5.5. The refined test profile can be used for further, more targeted test runs.

5.2 Ontology Structure

KGs have an internal structure that must be considered when designing the test profile: KGs (and in particular the ontologies they contain) are not built from scratch for each application but follow established ways to organize knowledge by importing ontologies that describe more general concepts.

The most general of these ontologies are called *top-level ontologies*, such as SUMO [65] or BFO [66]. They describe the most abstract terms. Often, an existing *mid-level ontology* is used, which captures the concepts that are relevant for the application domain. An example of such an ontology is the CORA ontology for robotics and automation [67]. Only the *bottom-level ontology* is developed specifically for the SUT and captures the classes but also individuals and relations that are relevant for the use case. As the top- and mid-level ontologies are usually imported, we assume that (i)

they contain some superfluous parts and (ii) they are correct w.r.t. the domain they describe. Hence, one often does not want to target them with the mutation operators. Instead, one wants to reveal information about the interaction of the software with the part of the KG that is specifically designed for the SUT. To achieve this, the user can (i) restrict the domain-specific mutation operators to only target the bottom-level ontology and (ii) use the robustness mask to ensure that all parts of the KG that are contained in the top- and mid-level ontologies are unaffected by the (domain-independent) mutation operators.

5.3 Mutation Operators to Ensure Input Feature Coverage

Deciding which mutation operators to include in the test profile can be a difficult task and heavily depends on the SUT. One guide is to select mutation operators that maximize the coverage of features that occur in the input KGs. It has been shown that a larger coverage of those features in test inputs leads to a larger code coverage of the SUTs [21]. It is thus desirable that the mutated KGs cover as many input features as possible.

In our case, the features are the graph patterns that are allowed to occur in the input, e.g., there can be a fixed set of IRIs that are allowed to be used a predicates and no other IRI is allowed to be used in this way. For every allowed graph pattern, there should be at least one mutation, or a combination of applying several mutations, that add this feature to the KG. E.g., in the case where we have a fixed set of IRIs allowed as predicates, there should be one mutation for each of those IRIs that adds a triple using this IRI as the predicate. This ensures, that all input features occur in mutant KGs, if enough KGs are generated, even if the features do not occur in the original KG.

We show in Section 6.3 that such a selection of the mutation operators really leads to a high input feature coverage in practice.

5.4 Automated Extraction of Mutation Operators

Designing mutation operators can be labor intensive. In particular, if the KGs in the domain of the SUT contain many features. We now propose an automated approach to extract mutation operators from reference KGs, which significantly reduces human effort in finding suitable mutation operators. Furthermore, the automated extraction reduces the dependency on domain expertise, which makes the operators more robust. The idea is that one can extract triple patterns that are common in KGs within the domain of interest and define mutation operators that either add such a pattern to the KG or delete such a pattern from the KG.

Extraction Method

We propose a two-stage process to extract mutation operators from reference KGs. First, we mine *association rules* from the KGs. Secondly, we extract mutation operators from the mined association rules.

Mining association rules from RDF KGs is a common task [68, 69]. In this work, we use the approach implemented by the tool RDFRules [12]. The goal is to mine

Horn rules that describe the structure of the RDF triples. A Horn rule has the form

$$B_1 \wedge \ldots \wedge B_n \rightarrow H$$

where $B_1 \wedge \ldots \wedge B_n$ is the body of the rule and H is the head of the rule. All atoms $B_1, \ldots, B_n$, and H are triples, which might contain variables. Let $\mathcal{B} = \{B_1, \ldots, B_n\}$ be the set containing all body atoms. The intuition for the rule is that if a set of triples in the KG matches the triples in the rule body, there must be a triple in the KG matching the head triple. While there exist different approaches for extracting association rules from KGs, they all require to set some parameters for the mining process, e.g., how often a pattern needs to occur in the KG to be extracted as a rule. Setting those parameters is the only step that involves human input in our proposed extraction method. After that, the rule mining process and also the extraction of the mutation operators is fully automatic.

Each obtained rule describes one pattern that occurs frequently in the reference KGs. We derive mutation operators that change KGs while preserving these patterns. For each rule, we derive two types of mutation operators: operators that add this pattern to the KG and operators that remove this pattern. Remember that a mutation operator is a pair (S, R), where S is the graph pattern that is searched for and R is the pattern that replaces the found pattern (see Section 4.1).

Addition. To construct a mutation operator that adds the pattern described by a rule, we first select a subset $\mathcal{B}'$ of the atoms in the rule body. The selected atoms are going to be searched for in the KG, while the rest of the body atoms are added as new triples to the KG together with the atom in the head. Formally, the mutation operator is a tuple (S, R) where $S = \mathcal{B}'$ and $R = \mathcal{B} \cup \{H\}$. We get the set of all such mutation operators by iterating over the possible subsets of body atoms $\mathcal{B}'$.
Removal. To construct a mutation operator that removes an occurrence of the rule pattern from the KG, a non-empty subset $\mathcal{B}'$ of atoms from the rule body is selected to be deleted. The operator selects a subgraph of the KG that contains all atoms from the body and the head of the rule. When the operator is applied, the triples in the selected subset are removed. Additionally, the head of the rule is removed because it depends on the atoms in the rule body, some of which were deleted. Formally, the mutation operator is a tuple (S, R) where $S = \mathcal{B} \cup \{H\}$ and $R = \mathcal{B} \setminus \mathcal{B}'$. We get the set of all such mutation operators by iterating over the non-empty subsets of body atoms $\mathcal{B}'$. We do not select the empty set because deleting the head atom while keeping all the body atoms would violate the pattern described by the rule.

Example

We consider our running example of an underwater robot inspecting a pipeline from Section 3. A mined association rule could be

$$(x \ \text{type} \ \text{Pipe}) \wedge (x \ \text{nextTo} \ y) \rightarrow (y \ \text{nextTo} \ x)$$

where x and y are variables. This rule states that if a node of type Pipe is related to another node by a nextTo relation, then this nextTo relation is symmetric.

To derive an operator that adds this pattern, we select a subset of the atoms in the rule body. For this example, we select $\mathcal{B}' = \{(x \,\text{type}\, \text{Pipe})\}$. The resulting operator is applicable to all nodes x in the KG that occur in a triple $(x \,\text{type}\, \text{Pipe})$. When the operator is applied, a fresh node is added to the KG to represent y, because y occurs in the rule but not in the selected atom. Furthermore, the operator adds the triples $(x \,\text{nextTo}\, y)$ and $(y \,\text{nextTo}\, x)$ to the KG, where the variables are replaced by the selected nodes.

To derive an operator that removes the pattern described by the rule, we select a non-empty set of body atoms. For this example, we again select $\mathcal{B}' = \{(x \,\text{type}\, \text{Pipe})\}$. The resulting operator is applicable to all nodes x and y that occur in the triples $(x \,\text{type}\, \text{Pipe}), (x \,\text{nextTo}\, y), (y \,\text{nextTo}\, x)$. When the operator is applied, the selected atom as well as the rule head is removed; i.e., the triples $(x \,\text{type}\, \text{Pipe})$ and $(y \,\text{nextTo}\, x)$ are removed from the KG, where the variables are replaced by the selected nodes.

Limitations

There are some limitations inherent to the proposed extraction approach.

- The approach is not fully automatic as some parameters for the rule mining process must be set. However, the approach significantly reduces the required manual effort compared to designing all mutation operators manually.
- The approach relies on having enough reference KGs to mine the rules from, i.e., we can only extract mutation operators for patterns that occur frequently.
- It is not possible to derive certain types of mutation operators because all variables in the association rules are universally quantified but existentially quantified variables can capture some patterns better. For example, the pattern that every robot is at one location can not be represented by an association rule.

Despite these limitations, we found the proposed approach to be useful in practice (see Section 6.3.7).

5.5 Obtaining Robustness Masks

As explained in Section 4, our testing architecture relies on the provision of a robustness mask to identify the mutants that should be used for testing. In general, such a mask does not exist when the SUT is first tested as it is usually not created together with the SUT.

Iterative Approach

We propose the following iterative approach to develop and refine a robustness mask over time. The user starts testing the SUT with an empty mask, i.e. all consistent mutants are considered for testing. If the test result complies with the oracle, the user generates a new mutant and repeats this process until a mutant is found for which the software's output does not match the oracle. When such a mutant is found, the mask needs to be updated. This requires a user that has some understanding about the SUT and the domain it operates in. The user needs to identify what part of the mutant, i.e. which mutation, causes the unexpected behavior. The mask is then updated to forbid such mutants in the future, i.e. such mutants do not conform to the updated

```
PipeStart
  a sh:NodeShape ;
  sh:property [
    sh:path :nextTo ;
    sh:maxCount 1 ;
  ].
```

(a)

```
:AUVStart
  a sh:NodeShape ;
  sh:targetNode :auv ;
  sh:property [
     sh:path :isAt ;
     sh:node :PipeStart ; ].
```

(b)

```
:LinearPipe
  a sh:NodeShape ;
  sh:targetClass :Pipe ;
  sh:property [
    sh:path :nextTo ;
    sh:maxCount 2 ;
  ].
```

(c)

```
:ConnectedPipe
  a sh:NodeShape ;
  sh:targetClass :Pipe ;
  sh:property [
    sh:path (:nextTo*)/(^isAt) ;
    sh:hasValue :auv ;
    ].
```

(d)

Fig. 6: Four SHACL shapes that are useful for the robustness mask of the pipeline inspection example.

mask. Afterwards, the process is repeated. This is done until a sufficiently accurate mask is found, e.g. because a large number of mutants can be generated without any deviations from the oracles.

Example

We demonstrate the iterative approach of developing a robustness mask using our running example from Section 3. Remember that the mutant in Figure 1d was an example for which the SUT does not work as expected. We assume that we found this mutant with the initial (empty) mask. Hence, we want to specify a new mask such that this KG does not conform to it. One example of such a mask would be the mask containing the two SHACL shapes in Figures 6a and 6b. The first shape specifies a node that has at most one `:nextTo` relation, i.e. a node that is at the beginning of a pipe structure and not in the middle. The second shape targets the node `:auv` and requires that this node is only linked to nodes defined by the first shape by an `:isAt` relation. Together, the shapes express that the node `:auv` is at a position that is at the beginning of the pipe structure. The mutated KG in Figure 1d does not conform to this mask as the segment `:p1`, which is the one where the AUV is, is in the middle of the pipe structure, i.e. this segment is connected to two other segments via a `:nextTo` relation.

Observe that the second version of the mask can still be further refined. It only filters out KGs where the branching of the pipe structure occurs at the segment where the AUV is located. Further generation of mutants using this second mask might generate the KG in Figure 7, for which the SUT does not inspect all pipe segments

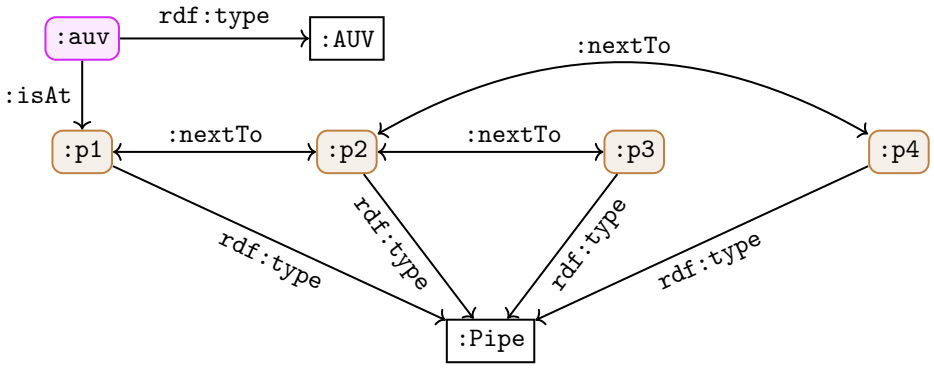


Fig. 7: A mutant of the KG in Figure 1c with two new pipe segments.

because the AUV does only inspect one of `:p3` and `:p4`. This mutant can be generated by applying the mutation that adds a new pipe segment twice. The mutant also hints at how to expand the mask: we need to add a shape that forbids the branching of the pipe structure as the algorithm only works on linear pipe structures. The SHACL shape in Figure 6c describes the desired structure. The shape specifies that every pipe segment is connected to at most two nodes via `:nextTo` relations. The mutant in Figure 7 does not conform to this shape as the node `:p2` has three nodes with which it is in a `:nextTo` relation. Hence, we obtain the third version of the mask by adding the shape from Figure 6c.

Although it eliminates many invalid input KGs, this third version of the mask is still not perfect. A fourth SHACL shape that needs to be added to the mask is shown in Figure 6d. It describes the connectivity of the pipe structure, i.e. that all pipe segments are connected to each other and can be reached from the `:auv`. To specify this requirement, we use a complex path condition, expressing a sequence of `:nextTo` relations of any length followed by an inverse `:isAt` relation.

Overall, the fourth mask for our example might contain all the shapes from Figure 6. Together with the shape that we introduced in Figure 1b, we finally obtain a robustness mask for our example that describes all requirements of the algorithm on the structure of the KG.

5.6 Interpretation of Results

Apart from identifying test cases for which the SUT behaves unexpectedly, i.e. the output differs from the oracle, the final obtained robustness mask is also a relevant result. The mask describes KGs for which the SUT behaves as expected. Hence, if the mask is large, the software component is less robust than if it only contains a few shapes. Furthermore, the mask describes the parts of the KG that are relevant for the SUT, i.e. it can help to identify parts of the KG that are superfluous. We expect such parts to occur rather often, as KGs are often not designed for a specific piece of software but rather for a domain and thus capture more knowledge than needed for the specific SUT. Additionally, the ontologies contained in the KG are often based on existing, more general ontologies, thus containing axioms that are not relevant.

5.7 Discussion

Two issues warrant some further discussion: Dealing with *redundant mutants* during generation, and the *human aspect* in using our methodology.

Redundancy

We consider two mutated input KGs as redundant if they unveil the same underlying fault, i.e., a bug in the SUT or an imprecision in the, yet incomplete, robustness mask. Redundant input mutants can be logically equivalent, but do not have to be. To avoid generating redundant input mutants, the robustness mask must be modified by adding shapes that prohibit the generation of an input mutant that triggers the same fault. In case the fault is an incomplete robustness mask, this is a special case of the overall iterative refinement. In case the fault is a program bug, this is akin to avoid fuzz blockers [70]. The mask not only prevents using the exact same mutant again for testing but defines equivalence between mutants on a program-semantic level.

Human Aspect

While test-case reduction and extraction of domain-specific mutation operators can be automated, our framework is still a black-box testing framework and consequently the iteration approach requires the user to refine the mask, as insights into the source code of the program are needed.

This can lead to the creation of different robustness masks. As a robustness mask is essentially a software specification, namely an interface specification, this is a special case of inconsistent specifications due to different views and skills of different stakeholders and users. In particular, different developers may have a different view about what is a bug of the program and what should be a forbidden mutation. Similarly, developers less familiar with semantic technologies may struggle to produce the right shapes to add and when the users understanding of the SUT is limited, there is the risk that generated mask is too strict, because our testing framework can only reveal that a mask is too permissive.

Inconsistent specification is a problem beyond our iterative approach, and we leave the applicability of common strategies to resolve this [71] for future work.

6 Evaluation

6.1 Research Questions

We recall our seven research questions that we aim to answer with this evaluation:

- **RQ1:** How can one *test the integration* of software with KGs?
- **RQ2:** Which *restrictions of the integration* of the KG and the software can be characterized using mutation operators and robustness masks?
- **RQ3:** What are the consequences of using *domain-specific mutation operators* compared to domain-independent mutation operators?
- **RQ4:** To what extent can mutation operators guarantee coverage of input features?
- **RQ5:** How much can input KGs be reduced in size to reveal root causes?

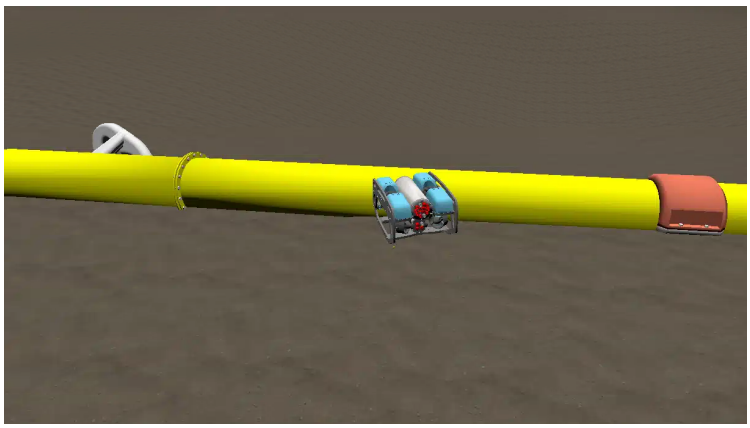


Fig. 8: Underwater robot simulation Suave with an AUV inspecting a pipeline.

- **RQ6:** Can an *iterative process* be used to develop a robustness mask that characterizes the interaction between the software and the KG?
- **RQ7:** To what extent can mutation operators be extracted from reference KGs?

6.2 Experimental Design and Setup

We apply our testing methodology to three different systems to answer these research questions. The three systems differ in many aspects, including the application domain, the way they access the KG and the relevant mutation operators, and thus allow us to derive a balanced view on our method.

The framework and the mutation operators are implemented using Kotlin. The implementation and data for reproducing our results can be found in the auxiliary material [72].

One domain expert from the field of KGs designed the mutation operators and refined the robustness masks. For the systems Suave and Geo, the developers where consulted to assist with the design of domain-specific mutation operators.

6.2.1 Testing Campaign 1: Suave

The first system is Suave [5]. It is a control system for autonomous underwater robots implemented on top of ROS2[73]. The mission of the robot is to find and inspect a pipeline on the sea floor. While doing so, the current and visibility of the water can change and thrusters of the robot can fail. This requires the robot to adapt its control algorithms, e.g. change the search pattern. The system that we are testing is responsible for this higher level of control called *metacontrol* [74]. The KG contains information for introspection, describing the components of the robot, how they interact and which capabilities are provided by which interactions. It also captures the relevant environment parameters, e.g. the water visibility, to derive estimations about the quality of different control algorithms. The ontology contained in the KG has three different levels: the top- and mid-level ontologies contain the axioms of the TOMASYS

Table 1: Implemented mutation operators. Operators marked with "*" are taken from Porn and Peres [30], the other operators are introduced by us.

name	description	target	domain-specific?	used in
CEUA*	remove one conjunct in a complex subclass axiom	TBox	no	Geo
CEUO*	remove one disjunct in a complex subclass axiom	Tbox	no	Geo
ACATO*	replace "and" with "or" in a subclass axiom	TBox	no	Geo
ACOTA*	replace "or" with "and" in a subclass axiom	TBox	no	Geo
ReplaceSibling	replace class by sibling class in a subclass axiom	TBox	no	Geo
ChangeDataProperty	change data property value to one that is in the domain of the property	ABox	no	Geo/ Suave
ChangeDouble	change value of a double	ABox / TBox	no	Geo
AddInstance	add a new instance of a class	ABox	no	Suave
RemoveTriple	delete a triple	ABox / TBox	no	Suave
AddRelation	add a relation between nodes	ABox	no	Suave
ChangeRelation	change object in a triple	ABox	no	Suave
RemoveNode	delete a node (including deleting it from all relations it occurs in)	ABox / TBox	no	Suave
AddThruster	add a new thruster to the robot	ABox	yes	Suave
AddQAEstimation	add new "quality-attribute estimation" relation	ABox	yes	Suave
RemoveQAEstimation	remove "quality-attribute estimation" relation	ABox	yes	Suave
ChangeSolvesFunction	change target of a "solves function" relation	ABox	yes	Suave
ChangeHasValue	change target of a "has value" relation to random decimal	ABox	yes	Suave
ChangeQAComparison	change target of a "qa comparison operator" relation to a different operator	ABox	yes	Suave

architecture [74], which describes self-adaptation. The low-level ontology contains the information about components, capabilities and restrictions of capabilities that are specific for the underwater inspection scenario.

Suave does not access the KG using queries, instead the nodes in the KG become Python objects and Python classes. This makes extracting the relevant parts of the KG with traditional methods, e.g. through analyzing modules of the ontology [75], impossible. Thus, Suave is a practical example where methods of black-box testing for the SUTs, like our method, are required.

Suave provides a realistic underwater physics simulation for the mission of inspecting a pipeline using the robot BlueRov2[5]. A screenshot of the simulation is shown in Figure 8. We use the simulation to decide if the metacontrol algorithm works correctly with a mutated KG. We classify finding and following the pipeline as "pass" and not being able to do so as "fail". Because the environmental parameters are chosen randomly by the simulation environment and differ for each test run, we run the simulation several times for each generated mutant. In our experiments, we used five test runs and the overall result is the majority of the results of the individual test runs.

```
@prefix mask:    <https://www.ifi.uio.no/tobiajoh/mask#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix schema:  <http://schema.org/> .
@prefix sh:      <http://www.w3.org/ns/shacl#> .
@prefix suave:   <http://www.metacontrol.org/suave#> .
@prefix tomasys: <http://metacontrol.org/tomasys#> .

mask:shape0  rdf:type  sh:NodeShape;
        sh:property    [ sh:hasValue  owl:NamedIndividual;
                         sh:minCount  1;
                         sh:path      rdf:type ];
        sh:targetNode  suave:qa_inspect_efficiency_high .

mask:shape1  rdf:type  sh:NodeShape;
        sh:property    [ sh:hasValue  1.0;
                         sh:minCount  1;
                         sh:path      tomasys:hasValue ];
        sh:targetNode  suave:qa_inspect_efficiency_high .

mask:shape2  rdf:type  sh:NodeShape;
        sh:property    [ sh:hasValue  owl:NamedIndividual;
                         sh:minCount  1;
                         sh:path      rdf:type ];
        sh:targetNode  suave:qa_search_efficiency_medium .

mask:shape3  rdf:type  sh:NodeShape;
        sh:property    [ sh:hasValue  <http://ros/mros#performance>;
                         sh:minCount  1;
                         sh:path      tomasys:isQAtype ];
        sh:targetNode  suave:qa_search_efficiency_medium .

mask:shape4  rdf:type  sh:NodeShape;
        sh:property    [ sh:hasValue  owl:NamedIndividual;
                         sh:minCount  1;
                         sh:path      rdf:type ];
        sh:targetNode  suave:qa_motion_efficiency_degraded .

mask:shape5  rdf:type  sh:NodeShape;
        sh:property    [ sh:hasValue  0.75;
                         sh:minCount  1;
                         sh:path      tomasys:hasValue ];
        sh:targetNode  suave:qa_motion_efficiency_degraded .

mask:shape6  rdf:type  sh:NodeShape;
        sh:property    [ sh:hasValue  owl:NamedIndividual;
                         sh:minCount  1;
                         sh:path      rdf:type ];
        sh:targetNode  suave:qa_search_efficiency_low .

mask:shape7  rdf:type  sh:NodeShape;
        sh:property    [ sh:hasValue  0.25;
                         sh:minCount  1;
                         sh:path      tomasys:hasValue ];
        sh:targetNode  suave:qa_search_efficiency_low .

mask:shape8  rdf:type  sh:NodeShape;
        sh:property    [ sh:hasValue  owl:Ontology;
                         sh:minCount  1;
                         sh:path      rdf:type ];
        sh:targetNode  <http://www.metacontrol.org/suave> .
```

Fig. 9: Example mask from Suave testing campaign.

Test Profile

We use 19 mutation operators for our evaluation of Suave, which are depicted in Table 1. We chose operators that fit our evaluation scenarios. About half of the operators target only ABox axioms, about a third of the operators target TBox axioms and the remaining operators can be applied to both ABox and TBox axioms. It is no coincidence that this difference is mirrored by the scenarios, in which the operators are used: in Geo, we only mutate the TBox and in Suave only the ABox. This is the case because the KG in Geo does not contain any ABox axioms that we can mutate. The KG for Suave does contain both types of axioms, TBox and ABox axioms, but the software depends so heavily on the TBox axioms, which are completely described in the top- and mid-level ontologies, that all modifications of the TBox axioms lead to failure. Therefore, we only mutate the ABox for Suave. By using these two systems for our evaluation, we already cover both types of mutations.

In general, all of our mutations target the low-level ontology as we are interested in analyzing the part of the KG that was developed specifically for the SUT and not the representations of general knowledge.

The mutation operators used for Suave target the description of the components and capabilities. This includes domain-independent operators that allow to add (and delete) individuals and relations between individuals. The domain-specific operators are mostly refinements of the domain-independent ones, e.g. they add or change one specific type of relation. For the changes, we choose relations that are most relevant for the scenario. An example of a more complex domain-specific mutation operator is "AddThruster", which involves adding a new node of class "Thruster" and adding several relations that include the new node to connect it properly to the existing configurations.

The mask for this testing campaign is not static but is refined during the experiments. We follow the process described in Section 5.5; i.e., we start with an empty mask for the initial test profile. Over all, we use eight different test profiles with eight different masks that share the same mutation operators. All masks can be found in the auxiliary material [72].

As an example, the mask that we obtain after the first refinement is shown in Figure 9. The mask consists of nine shapes and each shape ensures that one specific triple is present in the KG. I.e., the mask ensures that a sub-graph of the original KG with nine triples is not changed by the mutations. A shape ensures that a triple (s, p, o) is present in the KG by connecting the subject s via a `sh:targetNode` relation and representing p and o by a node that is connected via a `sh:property` relation. The relation `sh:path` connects to the property p and the relation `sh:hasValue` connects to the object o. Adding "1" via a `sh:minCount` relation ensures that the triple is contained. The names of the shapes are irrelevant and we thus use generic names.

We generated mutated KGs in batches and updated the robustness mask accordingly after obtaining the results for all the runs in a batch of size 10–30. Running one test case in Suave requires about 15 min, and overall 15% of the tested mutants led to failed test runs. With the empty mask, the failure rate was 50%.

[5] https://bluerobotics.com/store/rov/bluerov2/

6.2.2 Testing Campaign 2: Geo

The second system that we use for our evaluation is a geological simulator [6]. The KG describes an ontology of geological formations, processes and process triggers, and their relation. In particular, the ontology specifies triggers for oil maturation. The software simulates processes happening in the formations using the triggers described in the KG. The ontology is based on the top-level ontology BFO [66] and the mid-level ontology GeoCore [76]. The bottom-level ontology describes the geological processes and triggers relevant for the oil maturation simulation. The scenario, i.e. the types of formations and their layering, is described outside the KG.

To evaluate if a mutated KG preserves the correctness of the simulation, we track whether oil maturation occurs. The oracle, i.e. whether maturation is expected, is generated together with the scenario. As the simulation is deterministic, we only need to run the simulation once for each mutant.

Test Profile

The mutation operators used for Geo are shown in Table 1 and target complex subclass axioms, which are the key elements of the ontology describing the oil maturation triggers. Notably, we consider mutations to remove parts of the conditions under which maturation is triggered. As the ontology contained in the KG does not contain many axioms, it does not make sense to define domain-specific operators as the domain-independent operators can already only be applied in very few places.

Again, the robustness mask is not the same throughout the testing campaign but we start with an empty mask and update it after obtaining test results. Over all, we use two different masks, i.e., two different test profiles. All masks can be found in the auxiliary material [72].

We generated mutated KGs in batches of size 100. Running one test case in Geo requires 5 min, and again overall 15% of the tested mutants led to failed test cases. With the empty mask, the failure rate was 29%.

6.2.3 Testing Campaign 3: Reasoners

The third testing campaign targets three ontology reasoners for the EL-profile of OWL [77]: HermiT [7], Pellet [8] and ELK [9], which are the most popular OWL reasoners. For the EL-profile reasoners are expected to be mature enough to work on this fragment reliably [78]. The KG describes in this setting the ontology that is to be analyzed by the reasoners.

We tested the reasoners on two tasks: deciding the consistency of the ontology and computing the class hierarchy, i.e., all subclass relations that can be inferred from the input ontology. Those tasks are two of the most common ones for reasoners and also have been used in the past to evaluate the performance of OWL reasoner [78]. In addition to monitoring for exceptions, we used *differential testing* to evaluate the correctness of the output of the different tools. As the tools all rely on the same semantics for the KGs, the results should be equivalent. This type of oracle is relatively simple but quite effective, as we discuss in Section 6.3.

To get representative ontologies as seeds, which get mutated, we collected ontologies from the latest OWL Reasoner Competition [78]. We selected all KGs with ontologies in the EL-profile and then filtered to only use the KGs that are smaller than 1MB in file size, as the testing procedure gets slower, if the files are very large. This resulted in 307 KGs that we used as seeds.

For Reasoners, we did not generate KGs in batches nor did we develop a robustness mask as the reasoners are expected to run on all input ontologies. We used two sets of mutation operators: operators designed by domain experts and operators extracted from reference KGs. The following paragraph on the test profile details the selections of operators. We generated mutated KGs as input for the reasoners for 10 hours for each set of operators. We used all 307 seed KGs equally often for the generation.

For the operators designed by domain experts, this resulted in 1,991 test cases, of which 502 test runs resulted in the occurrence of an anomaly. For the extracted operators, this resulted in 4,057 test cases, of which 1,402 test runs resulted in the occurrence of an anomaly.

We manually sorted the test runs with anomalies into different classes based on the information in the anomaly reports. We defined four high-level classes. The first class contains input KGs that do not represent valid EL ontologies. The second class contains all test runs where an exception occurred. The remaining test runs were separated into two classes, depending on the type of reasoning task for which the tested reasoners differ. Either, the reasoners have different consistency evaluations or infer different class hierarchies. As a refinement step, we partitioned three of the four high-level classes into three subclasses each, based on the reasoner which is affected. For differences in reasoning results, this meant selecting the reasoner whose result did not conform with the other two reasoners. We did not partition the class of non-EL inputs as ordering by reasoners is not meaningful. Overall, this resulted in ten classes of test runs. Afterwards, we randomly selected one test case from each class and applied the test-case minimizer.

The resulting KGs are used to investigate the cause for the anomaly in detail. During this last step, we also applied some manual simplifications: (i) we renamed the nodes to give them generic names and (ii) we further removed triples from the KG. The latter resulted in not preserving the original outcome of the test run but if two reasoners differ on the axioms in the class hierarchy, we need to reduce the test case in such a way that they differ in at least one subclass relation and do not need to preserve all differences that occur when using the original test case.

Test profile

The mutation operators that are designed by domain experts are selected such that they address all allowed features of the EL-profile. For all allowed OWL declarations and axioms, we add at least one mutation that adds such a declaration or axiom and at least one mutation that deletes them. The only exception are annotations. Because annotations are not relevant for the reasoning process, we do not consider them for our testing. The operators are all chosen in a way that the ontology in the KG remains inside the EL-profile. The full list of all 55 used mutation operators can be found in Table 2. They target the TBox as well as the ABox.

Table 2: Mutation operators that are used to test Reasoners.

ID	mutation operation	target
1	add the declaration of new class	TBox
2	add the declaration of new object property	TBox
3	add the declaration of new data property	TBox
4	add a subclass relation between two classes	TBox
5	remove the subclass relation between two classes	TBox
6	add an equivalence relation between two classes	TBox
7	remove the equivalence relation between two classes	TBox
8	add a disjointedness relation between two classes	TBox
9	remove the disjointedness relation between two classes	TBox
10	replace a class with `owl:Thing`	TBox
11	replace a class with `owl:Nothing`	TBox
12	replace a class with a class with which it shares a super-class	TBox
13	declare an object property as reflexive	TBox
14	declare an object property as transitive	TBox
15	add a domain for an object property	TBox
16	add a domain for a data property	TBox
17	remove the domain of an object or a data property	TBox
18	add a range for an object property	TBox
19	add a range for a data property	TBox
21	remove the range of an object or adata property	TBox
22	add a sub-property relation between two object properties	TBox
23	add a sub-property relation between two data properties	TBox
24	remove a sub-property relation between two object or data properties	TBox
25	add an equivalence relation between two object properties	TBox
26	add an equivalence relation between two data properties	TBox
27	remove the equivalence relation between two object or data properties	TBox
28	add a sub-property relation containing a property chain	TBox
29	add a subclass axiom with a complex class expression containing an object intersection	TBox
31	add a subclass axiom with a complex class expression containing ObjectOneOf	TBox
32	add a subclass axiom with a complex class expression containing ObjectSomeValuesFrom	TBox
33	add a subclass axiom with a complex class expression containing ObjectHasValue	TBox
34	add a subclass axiom with a complex class expression containing ObjectHasSelf	TBox
35	add a subclass axiom with a complex class expression containing data type intersection	TBox
36	add a subclass axiom with a complex class expression containing DataHasValue	TBox
37	add a subclass axiom with a complex class expression containing DataOneOf	TBox
38	add a subclass axiom with a complex class expression containing DataSomeValuesFrom	TBox
39	remove one conjunct in complex subclass axiom (CEUA)	TBox
40	add datatype definition	TBox
41	add HasKey axiom	TBox
42	declare a new individual	ABox
43	remove an individual	ABox
44	add a class assertion for an individual	ABox
45	remove the class assertion of an individual	ABox
46	add an object property relation between two individuals	ABox
47	remove an object property relation between two individuals	ABox
48	add a negative object property relation between two individuals	ABox
49	remove a negative object or data property relation	ABox
50	add SameIndividual relation for two individuals	ABox
51	remove SameIndividual relation for two individuals	ABox
51	add DifferentIndividuals relation for two individuals	ABox
52	remove DifferentIndividuals relation for two individuals	ABox
53	add a data property relation from an individual	ABox
54	add a negative data property relation from an individual	ABox
55	remove a data property relation	ABox

The second set of mutation operators is extracted from the 307 KGs that we selected as seeds. We follow the process described in Section 5.4. We use the state-of-the-art tool RDFRules [12] to extract the association rules from the reference KGs. To use the tool RDFRules, we select four parameters: the number of occurrences of the rule body in the KG (50), the number of occurrences of the rule head in the KG (20), the minimum share of cases where the rule is satisfied (80%) and the maximal number of atoms in the rule (3). We selected these parameters to produce a decent amount of meaningful association rules based on some initial testing. Furthermore, we removed all rules that contain IRIs that are domain-specific. Thus, we only obtain association rules that are in general applicable to KGs representing EL ontologies. We extract the rules from each KG individually. In our experiments, the median number of extracted association rules from a KG is 4 while the minimum is 0 and the maximum is 26. We combine all extracted association rules into one set. The whole extraction process took about 6 minutes and produced 116 association rules. Using the process described in Section 5.4, we obtain 741 mutation operators from the association rules, which we use in the testing campaign.

We only use the empty mask for this testing campaign. Because running the reasoners is fast, i.e., only a few seconds, filtering the test inputs is less important. Defining a mask that captures exactly the KGs that represent valid EL ontologies is complex and the mutation operators already ensure that most mutants are within the domain of interest. The few mutants where this is not the case are detected later by the reasoners.

6.2.4 Evaluation Design for Analysis of Input Feature Coverage

We use the systems Suave and Reasoners to analyze what impact the selection of the mutation operators and the number of applied operators have on the coverage of input features. For both systems, this involves three steps:

1. Identification of the input features
2. Generation of mutants and analysis of covered input features
3. Using statistical tests on gathered data

1. Identification of the Input Features

For Suave, the input KGs are those KGs that represent the architecture of the Suave underwater robot using the TOMASYS architecture. Hence, we consider all IRIs that starts with one of the three domain-specific prefixed, i.e., mros, tomasys and suave, as the features. A list with all of them can be found in Table 3.

For Reasoners, the input KGs represent ontologies in the OWL-EL profile [77]. While there is no restriction on the data, i.e., the names of classes, properties or individuals, this restricts the IRIs that are allowed to express the OWL axioms. We manually identified all IRIs that are allowed (see the list of the IRIs in Table 4) using the definition of the OWL-EL profile [77] and the definition of how to translate OWL axioms to RDF triples [79].

2. Generation of Mutants and Analysis of Covered Input Features

We use 18 different numbers of mutation operators, ranging from 0, i.e., no mutation, to 100. We use the two sets of mutation operators for Suave (domain-specific

Table 3: All IRIs that describe features that occur in KGs of Suave. There are 88 in total. We use prefixes to shorten the notation: "suave:" for "http://www.metacontrol.org/suave#", "tomasys:" for "http://metacontrol.org/tomasys#" and "mros:" for "http://ros/mros#".

mros:requiredBy	tomasys:QAvalue	tomasys:typeF
mros:performance	tomasys:hasValue	tomasys:typeFD
mros:o	tomasys:isQAtype	tomasys:typeC
mros:c	tomasys:Function	tomasys:FunctionGrounding
mros:fg	tomasys:Component	tomasys:o_always_improve
mros:safety	tomasys:Objective	tomasys:binding_component
mros:energy	tomasys:fd_error_log	tomasys:cspec_availability
mros:qa	tomasys:FunctionDesign	tomasys:QualityAttributeType
mros:nfrv	tomasys:solvesF	suave:f_follow_pipeline
mros:nfr	tomasys:hasQAestimation	suave:fd_all_thrusters
mros:fd	tomasys:requiresO	suave:obs_water_visibility
mros:qav	tomasys:Binding	suave:water_visibility
tomasys:c	tomasys:hasQAvalue	suave:fd_spiral_high
tomasys:cs	tomasys:hasBindings	suave:f_generate_search_path
tomasys:fg	tomasys:ComponentClass	suave:qa_water_visibility_high
tomasys:o	tomasys:ComponentState	suave:f_maintain_motion
tomasys:cc	tomasys:fd_efficacy	suave:fd_spiral_medium
tomasys:b	tomasys:solvesO	suave:fd_recover_thrusters
tomasys:fd	tomasys:cc_availability	suave:fd_follow_pipeline
tomasys:r	tomasys:qa_critical	suave:fd_spiral_low
tomasys:roles	tomasys:hasNFR	suave:eqa
tomasys:Role	tomasys:roleDef	suave:fd
tomasys:fg_status	tomasys:binding_role	suave:eqav
tomasys:c_status	tomasys:fd_realisability	suave:mqa
tomasys:o_status	tomasys:o_updatable	suave:fg
tomasys:b_status	tomasys:cc_unique	suave:mqav
tomasys:qa_comparison_operator		tomasys:ComponentSpecification
suave:qa_search_efficiency_high		suave:qa_search_efficiency_low
suave:qa_motion_efficiency_normal		suave:qa_water_visibility_medium
suave:qa_search_efficiency_medium		suave:qa_inspect_efficiency_high
suave:qa_motion_efficiency_degraded		

and domain-independent) and three sets of mutation operators for Reasoners (domain-specific, domain-independent and extracted) as described in the previous sections about the test profiles. For each set of mutation operators and each number of applied mutation operators, we generate 100 mutants for Suave and 500 mutants for Reasoners. We generate more mutants for Reasoners because the coverage of the mutants varies more and we want to get significant results. Having 18 numbers of applied mutation operators and 2 (or 3) sets of mutation operators, this results in $18 \cdot 2 = 36$ (or $18 \cdot 3 = 54$) setups for which we generate mutants. We measure for each mutant which of the input features are covered.

The number of mutants used for testing has an impact on the covered features. To evaluate this aspect, we accumulate the features of 10 (and 100) mutants. We consider 100 random samples of 10 (and 100) mutants from the generated mutants for each setup.

Table 4: All IRIs that describe features that are allowed in the EL-profile of OWL. There are 64 in total.

rdf:type	xsd:base64Binary	owl:equivalentClass
rdf:PlainLiteral	xsd:anyURI	owl:disjointWith
rdf:XMLLiteral	xsd:dateTime	owl:AllDisjointClasses
rdf:first	xsd:dateTimeStamp	owl:members
rdf:rest	xsd:boolean	owl:propertyChainAxiom
rdf:nil	owl:real	owl:ReflexiveProperty
rdfs:Literal	owl:rational	owl:TransitiveProperty
rdfs:subClassOf	owl:Thing	owl:hasKey
rdfs:subPropertyOf	owl:Nothing	owl:sameAs
rdfs:domain	owl:Ontology	owl:differentFrom
rdfs:range	owl:imports	owl:AllDifferent
rdfs:Datatype	owl:namedIndividual	owl:NegativePropertyAssertion
xsd:decimal	owl:ObjectProperty	owl:sourceIndividual
xsd:integer	owl:DatatypeProperty	owl:assertionProperty
xsd:nonNegativeInteger	owl:AnnotationProperty	owl:targetIndividual
xsd:string	owl:Class	owl:targetValue
xsd:normalizedString	owl:intersectionOf	owl:Annotation
xsd:token	owl:oneOf	owl:annotatedSource
xsd:Name	owl:Restriction	owl:annotatedProperty
xsd:NCName	owl:onProperty	owl:annotatedTarget
xsd:NMTOKEN	owl:someValuesFrom	owl:hasValue
xsd:hexBinary		

3. Using Statistical Tests on Gathered Data

For each setup, we compare the distribution of shares of features covered by the mutants with the distribution of shares of features covered by the seed KGs. We use t-tests to decide if the mutants cover significantly more features than the seed KGs. We also compute the 95%-confidence interval for the difference of the means of the distributions.

For Suave, use a one-sample, two-sided t-test. We use a *one-sample* t-test because there is only a single seed KG, i.e., a single value to compare against. We do not perform tests for the set of domain-specific operators for Suave because nearly all of the mutants generated with them cover all features and no test is possible if nearly all values in a sample are the same.

For Reasoners, we use Welch's two-sample, two-sided t-test. We use *Welch's* t-test, because the variances of the distributions differ.

We interpret two results of the t-tests: the p-value and the 95%-confidence interval for the difference of the mean values. We use a limit of 0.05 for the p-value, i.e., if $p < 0.05$, we conclude that the feature coverage of the mutants is significantly different than the feature coverage of the seed KGs. We use the 95%-confidence interval to estimate the share of features that is additionally covered by the mutants. E.g., if the confidence interval is $[0.2, 0.4]$, we know that the difference between the mean share of features covered by the mutants is likely between 20% and 40% higher than the mean share of features covered by the seed KGs.

6.3 Results

For each research question, we provide our findings with the detailed insights that we gained from analyzing Suave, Geo and Reasoners. At the end of each paragraph we state a general answer of the respective research question.

6.3.1 RQ1: How can one test the integration of software with KGs?

Using mutations of the original KGs, we identified KGs for which the SUT does not behave as intended for all systems, Suave, Geo and Reasoners, with reasonable effort. The developed masks provide a declarative description of the restrictions, which can be interpreted by KG experts. We developed such a mask for the systems Suave and Geo.

Answer to RQ1: Our presented approach is well suited for integration testing of software with KGs.

6.3.2 RQ2: Which restrictions of the integration of the KG and the software can be characterized using mutation operators and robustness masks?

We identified three types of insight that can be very well detected: (i) parts of the KG that are superfluous, (ii) axioms that are missing in the KG and (ii) KGs that trigger a bug in the tested systems.

Superfluous Parts of KGs

Superfluous parts are to be expected in many KGs as they are often developed for more than one particular software. Depending on whether the superfluous part is irrelevant to solve the problem, this can indicate one of two modeling errors: (i) the model is unnecessarily large or (ii) a lack of separation of concerns. We found examples of both in our analyzed systems. Superfluous parts are detected, when we can apply mutation operators without any effect on the behavior of the SUT. We identified several complex subclass-axioms in the KG of Geo that had no effect on the behavior and thus describe relations that are not necessary for the intended scenarios. An example for the second modeling error was discovered using mutation "AddThruster" for Suave. Although the number of thrusters is relevant for the control algorithm of the robot, e.g. the number of thrusters is an upper limit of the degrees of freedom in which the robot can move, the software component does not rely on this information from the KG. This reveals a lack of separation of concerns in the system as this information has to be encoded in the software component and the KG simultaneously.

Missing Axioms in KGs

Our method can also discover axioms that are missing in the KG by generating graphs that are consistent w.r.t. the TBox in the KG but that do not conform to with what the software component expects. For example, we discovered that one of the data relations in Geo had no range associated with it. This allowed mutations where the data value was changed from a double to a boolean value, which the software could not handle. The solution is to add an axiom that specifies the correct range of the relation.

Table 5: All newly found bugs when testing Reasoners. The issueId refers to the id in the corresponding online issue tracker.

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 $\bot \sqsubseteq \top$

Bugs in Tested Systems

Lastly, our method also proved to be useful to find bugs in the SUTs. We found 6 previously unknown bugs in the OWL-EL reasoners that we tested. We found all the bugs in the campaign using the domain-specific operators designed by experts. We found the bugs P1 and P4 in the campaign using the extracted operators. Note, that we also found in both campaigns several test inputs that triggered a bug that was only recently discovered [13, bug O4]. We do not discuss this bug here as it is already documented in the mentioned publication and reported online.[6] Table 5 shows all bugs that we discovered in our testing campaign. Most of the bugs are exceptions that occur although the test KGs represent valid ontologies. One of the bugs (P1) only occurs non-deterministically, i.e., not every time the reasoner is called with the test input. Such bugs are particularly hard to describe as their occurrence is unpredictable. Two of the bugs are completeness bugs, i.e., the reasoner misses a subclass relation that should be contained in the class hierarchy. These bugs are especially important to find as there is no warning for the user that the computed result of the tool is incorrect. Such bugs are also known as *logical bugs*. We reported the found bugs in the corresponding issue tracker, if one exists. Detailed descriptions for all the discovered bugs can also be found in our supplementary material [72]. In the following, we discuss the three bugs where we could minimize the input sufficiently to analyze the trigger of the bug, i.e., P2, P4 and H1.

The bug with id P2 is one of the completeness bugs that we discovered, i.e., the computed result of the Pellet reasoner is logically incorrect. In particular, the reasoner misses a subclass relationship that can be inferred. Figure 10 shows the minimized mutant KG that triggers the bug. The KG represents two logical axioms: (i) that `:p` is a reflexive property and (ii) that `:B` is equivalent to the class of nodes that have an outgoing `:p`-relation to a node that is in class `:A`. Because `:p` is reflexive, every node

[6] https://github.com/Galigator/openllet/issues/89

```
@prefix : <http://example.org#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

:p rdf:type owl:ObjectProperty ,
            owl:ReflexiveProperty .

:A rdf:type owl:Class .

:B rdf:type owl:Class ;
   owl:equivalentClass [ rdf:type owl:Restriction ;
                         owl:onProperty :p ;
                         owl:someValuesFrom :A
                       ] .
```

Fig. 10: Mutant KG that triggers P2. The KG was automatically and afterwards manually minimized.

in `:A` has an `:p`-relation to itself, and hence every node in `:A` is also in `:B`. Pellet does not infer that `:A` is a subclass of `:B`, although it follows from the KG and thus we discovered a completeness bug.

The bug with id P5 is another completeness bug where Pellet misses a subclass relationship that can be inferred. Figure 11 shows the minimized mutant KG that triggers the bug. The KG represents two subclass axioms: (i) the class `:A` is a subclass of the nodes that have a `:p`-relation to a literal of type `xsd:normalizedString` and (ii) the set of nodes that have a `:p`-relation to a literal of type `xsd:string` is a subclass of the class `:B`. The datatype `xsd:normalizedString` is defined as strings without any characters representing white space [80] and hence is derived from `xsd:String`. Therefore, the restriction in the first axiom in the KG describes a subclass of the restriction in the second axiom in the KG. Thus, one can infer that `:A` is a subclass of `:B` but Pellet misses this inference. This is probably due to an incorrect treatment of the relationship between the datatypes.

The bug H1 is the only new bug that we found in the HermiT reasoner. The bug is that HermiT throws an exception when instantiated with the (already minimized) KG in Figure 12. The KG contains only one axiom: that the class of all things is a subclass of the empty class. This triple is a standard representation of a simple inconsistent axiom. Surprisingly, a KG containing this triple leads to an exception of the reasoner, instead of the reasoner simply reporting the inconsistency.

Answer to RQ2: We identified three types of restrictions of the interactions: (i) KGs that trigger a bug in the tested systems, (ii) parts of the KGs that are superfluous and (iii) axioms that are missing in the KGs.

```
@prefix : <http://www.example.org#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:p rdf:type owl:DatatypeProperty .

:A rdf:type owl:Class ;
   rdfs:subClassOf [ rdf:type owl:Restriction ;
                     owl:onProperty :p ;
                     owl:someValuesFrom xsd:normalizedString
                   ] .

:B rdf:type owl:Class .

[ rdf:type owl:Restriction ;
  owl:onProperty :p ;
  owl:someValuesFrom xsd:string ;
  rdfs:subClassOf :B
] .
```

Fig. 11: Mutant KG that triggers P5. The KG was automatically and afterwards manually minimized.

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

owl:Thing rdfs:subClassOf owl:Nothing .
```

Fig. 12: Mutant KG that triggers H1. The KG was automatically and afterward manually minimized.

6.3.3 RQ3: What are the consequences of using domain-specific mutation operators compared to domain-independent mutation operators?

In general, the usage of domain-specific mutation operators leads to less mutants that do not comply with the specified robustness mask or are inconsistent, as shown in Figure 13 for Suave. The figure also shows that this difference is independent of the number of mutation operators that are applied. Notably, the more mutation operators get applied, the greater the advantage of using domain-specific operators becomes. Hence, by using domain-specific operators, one can generate mutated KGs faster.

Similarly, the domain-specific mutation operators lead faster to test cases for which the SUT does not work correctly; i.e., the probability that a test run fails is higher for the same number of mutations. This is desirable, as failing test runs are our main

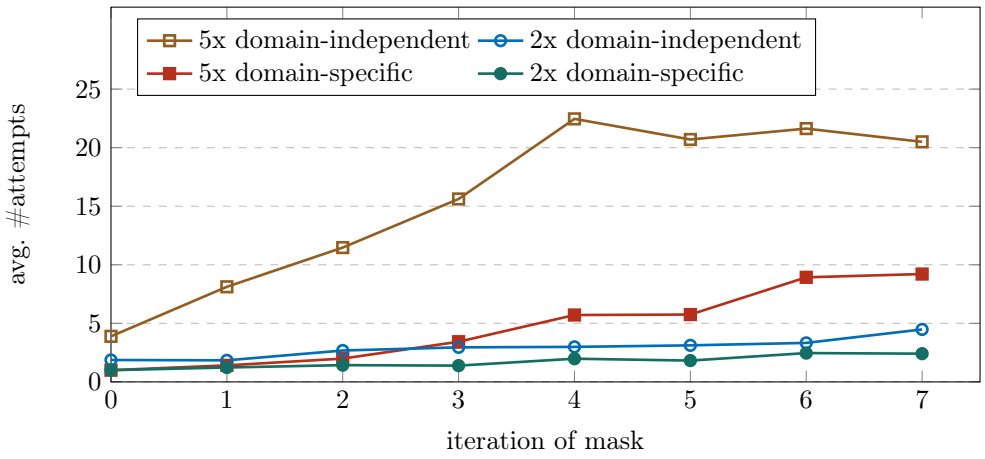


Fig. 13: Average number of attempts to create a valid mutant for each iteration of the mask for Suave. For each data point, batches of 100 mutants were created by applying two or five mutation operators respectively. The operators are either only domain-specific or only domain-independent mutation operators.

source to gain insight into the behavior of the SUT. Overall, it allows us to use less mutations to generate a test case, which is beneficial for developing a mask (see **RQ6**).

Answer to RQ3: domain-specific mutation operators behave differently from domain-independent ones: using domain-specific operators increases the probability to generate mutants that are valid and the generated mutants are more likely to lead to failing test runs.

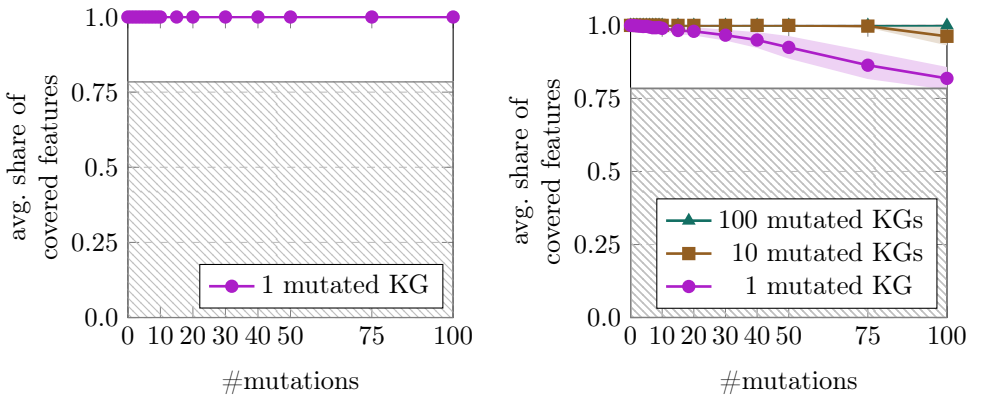
6.3.4 RQ4: To what extent can mutation operators guarantee coverage of input features?

We evaluate how much input features are covered when applying mutation operators using Suave and Reasoners. We chose these SUTs because they have a defined set of input KGs. To come up with a measurement of the input feature coverage, we first identify for both cases the IRIs, i.e., the terms, that form the KGs in their domains. For a given KG, we count how many of those IRIs occur and use that a the measurement for the input feature coverage.

To evaluate the impact of the selection of the mutation operators, we use a selection of domain-independent operators for comparison. We select the most general operators we defined, which add and delete nodes and triples. Namely, we select the operators *AddInstance*, *RemoveNode*, *AddRelation*, *ChangeRelation* and *RemoveNode* (see Table 1).

Input Feature Coverage for Suave

Figure 14 shows the impact of applying the mutation operators on the input feature coverage. We do not use a mask but consider all generated mutants. For the plot on



(a) Domain-specific operators (b) Domain-independent operators

Fig. 14: Input feature coverage for Suave. The plots show the share of covered features and the striped areas mark the share of features contained in the unmutatable part of the KG. For each number of mutations, 100 mutants where generated. For plot (a), domain-specific mutation operators are used. For plot (b), domain-independent mutation operators are used. The standard deviations are depicted as shaded areas. For most of the curves, they are smaller than 1% and hence not visible in the diagrams.

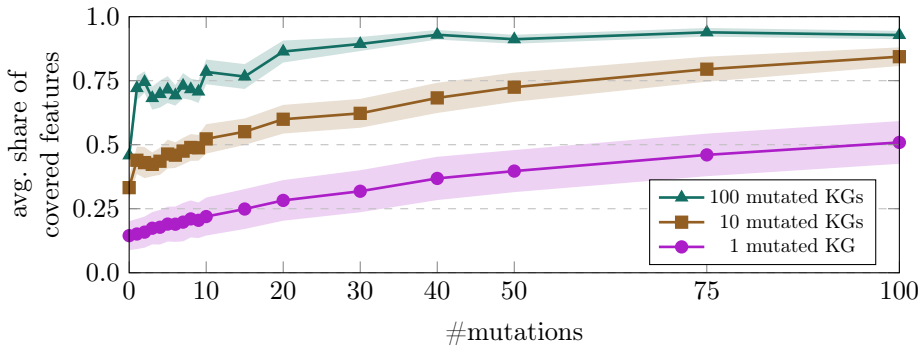
the left, we only use domain-specific mutation operators. For the plot on the right, we only use domain-independent mutation operators. The results of the t-tests are shown in Table 6.

Applying domain-specific mutation operators does not change the coverage of features notably. The main reason is that most features occur in the TBox of the contained ontology but the domain-specific operators only target the ABox. Therefore, only in a few cases do the mutated KGs not cover all input features. Another reason is, that most features (69 of 88) do occur in the part of the KG that is not allowed to be mutated by the mutation operators and thus guaranteeing that those features occur in every mutant KG.

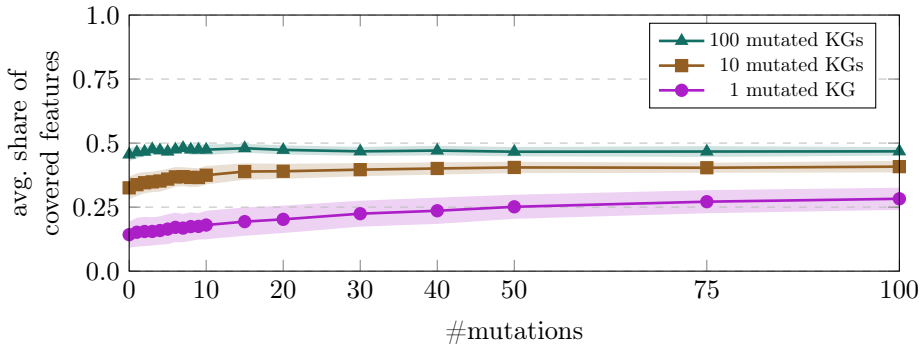
Applying domain-independent mutation operators generates test cases that do not cover all input features. This can be seen by the falling curve in Figure 14 (b) and by the negative values in the corresponding confidence intervals in Table 6. The table also shows that the differences are significant. This behavior is caused by the mutation operators that delete nodes and triples, which can lead to features being removed from the KG entirely. However, one can still cover all input features by generating sets of 10 or 100 test cases. This can be seen in Table 6 as nearly no test are possible for sets of 10 or 100 mutants because the sets of mutants contain all input features.

Input Feature Coverage for Reasoners

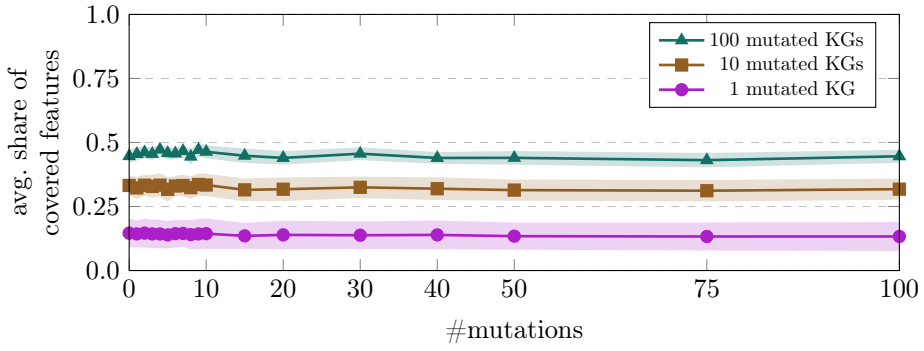
Figure 15 shows how many of the input features are covered by the mutants that we generate. Plot (a) shows the effect of using domain-specific mutation operators, plot (b)



(a) Domain-specific mutation operators



(b) Extracted operators



(c) Domain-independent operators

Fig. 15: Input feature coverage for Reasoners. Features are the IRIs that are allowed to express structural information in KGs containing ontologies in the EL profile of OWL. Plot (a) uses domain-specific mutation operators, plot (b) uses extracted operators, and plot (c) uses domain-independent operators. Curves show average share of covered features when considering a single test case or when accumulating features of 10 or 100 test cases. For each number of mutations, 500 mutants where generated. Shaded areas show standard deviations.

Table 6: Results for one-sample, two-sided t-tests for Suave using domain-independent operators (see Figure 14 (b)). Shares of features covered by mutants are compared to share of features covered by seed KG. Significant p-values (<0.05) are italic. Confidence Intervals show differences between mean feature coverage of mutants and seed KG. NA describes that no test is applicable because all samples of this collection of mutants contains all features.

#mutations	1 mutated KG		10 mutated KGs		100 mutated KGs	
	p-value	95%-CI	p-value	95%-CI	p-value	95%-CI
1,...,10	*<0.01*	[-0.02, 0.00]	NA	NA	NA	NA
15	*<0.01*	[-0.03, -0.01]	NA	NA	NA	NA
20	*<0.01*	[-0.03, -0.01]	NA	NA	NA	NA
30	*<0.01*	[-0.04, -0.02]	NA	NA	NA	NA
40	*<0.01*	[-0.06, -0.04]	NA	NA	NA	NA
50	*<0.01*	[-0.09, -0.06]	NA	NA	NA	NA
75	*<0.01*	[-0.15, -0.12]	*<0.01*	[-0.01, 0.00]	NA	NA
100	*<0.01*	[-0.19, -0.17]	*<0.01*	[-0.05, -0.03]	NA	NA

Table 7: Results for Welch's two-sample, two-sided t-tests for Reasoners using domain-specific operators (see Figure 15(a)). Shares of features covered by mutants are compared to shares of features covered by seed KGs. Significant p-values (<0.05) are italic. Confidence Intervals show differences between mean feature coverage of mutants and seed KGs.

#mutations	1 mutated KG		10 mutated KGs		100 mutated KGs	
	p-value	95%-CI	p-value	95%-CI	p-value	95%-CI
1	0.07	[-0.01, 0.02]	*<0.01*	[0.10, 0.12]	*<0.01*	[0.25, 0.27]
2	*<0.01*	[0.00, 0.03]	*<0.01*	[0.09, 0.11]	*<0.01*	[0.28, 0.30]
3	*<0.01*	[0.02, 0.04]	*<0.01*	[0.08, 0.10]	*<0.01*	[0.22, 0.23]
4	*<0.01*	[0.02, 0.05]	*<0.01*	[0.09, 0.11]	*<0.01*	[0.23, 0.25]
5	*<0.01*	[0.03, 0.06]	*<0.01*	[0.12, 0.14]	*<0.01*	[0.25, 0.27]
6	*<0.01*	[0.03, 0.06]	*<0.01*	[0.12, 0.14]	*<0.01*	[0.23, 0.24]
7	*<0.01*	[0.04, 0.06]	*<0.01*	[0.13, 0.15]	*<0.01*	[0.26, 0.28]
8	*<0.01*	[0.05, 0.08]	*<0.01*	[0.15, 0.17]	*<0.01*	[0.25, 0.27]
9	*<0.01*	[0.05, 0.07]	*<0.01*	[0.14, 0.17]	*<0.01*	[0.24, 0.26]
10	*<0.01*	[0.06, 0.08]	*<0.01*	[0.18, 0.20]	*<0.01*	[0.32, 0.24]
15	*<0.01*	[0.09, 0.12]	*<0.01*	[0.21, 0.23]	*<0.01*	[0.30, 0.22]
20	*<0.01*	[0.12, 0.15]	*<0.01*	[0.26, 0.28]	*<0.01*	[0.40, 0.41]
30	*<0.01*	[0.16, 0.19]	*<0.01*	[0.28, 0.30]	*<0.01*	[0.43, 0.44]
40	*<0.01*	[0.21, 0.24]	*<0.01*	[0.34, 0.36]	*<0.01*	[0.46, 0.48]
50	*<0.01*	[0.24, 0.27]	*<0.01*	[0.38, 0.40]	*<0.01*	[0.45, 0.46]
75	*<0.01*	[0.30, 0.33]	*<0.01*	[0.45, 0.47]	*<0.01*	[0.47, 0.49]
100	*<0.01*	[0.35, 0.38]	*<0.01*	[0.50, 0.52]	*<0.01*	[0.46, 0.48]

shows the effect of using operators extracted from reference graphs, and plot (c) shows the effect of only using domain-independent mutation operators as a baseline. The results of the t-tests are shown in Tables 7 to 9.

As we can see in Figure 15, the original KGs do not cover much of the input features. On average, a single KG only covers 15% of the input features and even 100

Table 8: Results for Welch's two-sample, two-sided t-tests for Reasoners using extracted operators (see Figure 15(b)). Shares of features covered by mutants are compared to shares of features covered by seed KGs. Significant p-values (<0.05) are italic. Confidence Intervals show differences between mean feature coverage of mutants and seed KGs.

#mutations	1 mutated KG		10 mutated KGs		100 mutated KGs	
	p-value	95%-CI	p-value	95%-CI	p-value	95%-CI
1	*<0.01*	[0.00, 0.02]	*<0.01*	[0.00, 0.02]	*<0.01*	[0.00, 0.02]
2	*<0.01*	[0.00, 0.02]	*<0.01*	[0.01, 0.03]	*<0.01*	[0.00, 0.02]
3	*<0.01*	[0.00, 0.02]	*<0.01*	[0.01, 0.03]	*<0.01*	[0.01, 0.03]
4	*<0.01*	[0.00, 0.03]	*<0.01*	[0.02, 0.04]	*<0.01*	[0.01, 0.02]
5	*<0.01*	[0.01, 0.03]	*<0.01*	[0.02, 0.04]	*<0.01*	[0.00, 0.02]
6	*<0.01*	[0.02, 0.04]	*<0.01*	[0.03, 0.05]	*<0.01*	[0.01, 0.03]
7	*<0.01*	[0.01, 0.04]	*<0.01*	[0.03, 0.05]	*<0.01*	[0.02, 0.03]
8	*<0.01*	[0.02, 0.04]	*<0.01*	[0.03, 0.05]	*<0.01*	[0.01, 0.03]
9	*<0.01*	[0.02, 0.04]	*<0.01*	[0.03, 0.05]	*<0.01*	[0.01, 0.03]
10	*<0.01*	[0.03, 0.05]	*<0.01*	[0.04, 0.06]	*<0.01*	[0.01, 0.03]
15	*<0.01*	[0.04, 0.06]	*<0.01*	[0.05, 0.07]	*<0.01*	[0.02, 0.03]
20	*<0.01*	[0.05, 0.07]	*<0.01*	[0.06, 0.07]	*<0.01*	[0.01, 0.03]
30	*<0.01*	[0.07, 0.09]	*<0.01*	[0.06, 0.08]	*<0.01*	[0.00, 0.02]
40	*<0.01*	[0.08, 0.10]	*<0.01*	[0.07, 0.09]	*<0.01*	[0.01, 0.02]
50	*<0.01*	[0.10, 0.12]	*<0.01*	[0.07, 0.09]	*<0.01*	[0.00, 0.02]
75	*<0.01*	[0.12, 0.14]	*<0.01*	[0.07, 0.09]	*<0.01*	[0.00, 0.02]
100	*<0.01*	[0.13, 0.15]	*<0.01*	[0.07, 0.09]	*<0.01*	[0.00, 0.02]

Table 9: Results for Welch's two-sample, two-sided t-tests for Reasoners using domain-independent operators (see Figure 15(c)). Shares of features covered by mutants are compared to shares of features covered by seed KGs. Significant p-values (<0.05) are italic. Confidence Intervals show differences between mean feature coverage of mutants and KGs.

#mutations	1 mutated KG		10 mutated KGs		100 mutated KGs	
	p-value	95%-CI	p-value	95%-CI	p-value	95%-CI
1	0.39	[-0.01, 0.01]	*<0.01*	[-0.02, 0.00]	*<0.01*	[0.00, 0.02]
2	0.96	[-0.01, 0.01]	0.71	[-0.01, 0.01]	*<0.01*	[0.01, 0.02]
3	0.42	[-0.01, 0.01]	0.16	[-0.01, 0.01]	*<0.01*	[0.00, 0.02]
4	0.39	[-0.02, 0.01]	0.64	[-0.01, 0.01]	*<0.01*	[0.02, 0.03]
5	0.05	[-0.02, 0.01]	*<0.01*	[-0.03, 0.00]	*<0.01*	[0.00, 0.02]
6	0.60	[-0.01, 0.01]	0.39	[-0.01, 0.01]	*<0.01*	[0.00, 0.02]
7	0.77	[-0.01, 0.01]	0.92	[-0.01, 0.01]	*<0.01*	[0.01, 0.03]
8	0.11	[-0.02, 0.01]	*0.01*	[-0.02, 0.00]	0.28	[-0.01, 0.01]
9	0.41	[-0.02, 0.01]	0.14	[-0.01, 0.01]	*<0.01*	[0.02, 0.03]
10	0.61	[-0.01, 0.01]	0.52	[-0.01, 0.01]	*<0.01*	[0.01, 0.03]
15	*0.02*	[-0.02, 0.00]	*<0.01*	[-0.03, 0.00]	0.36	[-0.01, 0.01]
20	0.06	[-0.02, 0.01]	*<0.01*	[-0.03, 0.00]	*<0.01*	[-0.02, 0.00]
30	*0.02*	[-0.02, 0.00]	*<0.01*	[-0.02, 0.00]	*<0.01*	[0.00, 0.02]
40	0.06	[-0.02, 0.01]	*<0.01*	[-0.02, 0.00]	*<0.01*	[-0.02, 0.00]
50	*<0.01*	[-0.02, 0.00]	*<0.01*	[-0.03, 0.00]	*<0.01*	[-0.01, 0.00]
75	*<0.01*	[-0.02, 0.00]	*<0.01*	[-0.03, 0.00]	*<0.01*	[-0.02, -0.01]
100	*<0.01*	[-0.02, 0.00]	*<0.01*	[-0.02, 0.00]	0.71	[-0.01, 0.01]

KGs together only cover 48% of the input features. In other words: half of the input features are never considered for testing.

For the domain-specific mutation operators, the rising curves in Figure 15 (a) show that the more mutations get applied, the more of the input features are covered. Our test results in Table 7 show that the difference to the seed KGs is statistically significant and the increase of the curves is confirmed by the increasing values of the confidence intervals. When using about 50 mutants the share of covered input features when considering 100 mutants plateaus at about 94%. The remaining features that are never used are related to annotations and as we omitted mutation operators to introduce annotations, those never get used.

For the extracted mutation operators, the curves in Figure 15 (b) rise slower than for the domain-specific mutation operators. This can also be seen in the smaller values of the confidence intervals in Table 8 compared to Table 7. Crucially, the feature coverage of the whole test suite, i.e., the curve representing 100 mutants, does not rise when more mutation operators are applied. Although the p-values in Table 8 show a significant difference to the seed KGs, the confidence intervals are very close to zero and thus the increase is not of practical value. This is to be expected. Features that do not occur in any reference KG are not introduced by the mutation operators because the mutation operators are extracted from the reference KGs. Nevertheless, the average number of covered features for an individual mutant KG increases when the mutation operators are applied, e.g., it increases to 28% when 100 operators are applied.

For the domain-independent mutation operators, the curves in Figure 15 (c) are nearly flat. We see the same trend in Table 9, where many distributions to not differ significantly from the distributions associated with the seed KGs and the confidence intervals indicate only very small differences between the features covered by the mutants and by the seed KGs. This shows that these operators do not increase the coverage of the input features.

Overall, we see that using a selection of mutation operators that is designed to maximize the feature coverage can indeed ensure that the input feature space gets covered when a reasonable numbers of mutations are applied and a decent number of mutants is generated. Extracted operators can also increase the coverage but to a lesser extent.

Answer to RQ4: Mutation operators can increase the coverage of input features in the mutants. Repeated application of mutation operators can lead to full coverage.

6.3.5 RQ5: How much can input KGs be reduced in size to reveal root causes?

To evaluate, how well our test-case minimizer works, we tried to minimize the KGs that triggered bugs while testing Reasoners. We chose these KGs, as they are rather large (several hundred triples) and thus make it very challenging to find the root cause of a bug. In total, we had 12 KGs to minimize. Some of the KGs trigger the same bugs. Our minimization algorithm is not applicable to two of the KGs. Bug P1 occurred only non-deterministically and hence the cause can not be reliably detected. One other KG

Table 10: Automatic reduction of the KGs that triggered anomalies in Reasoners. Some KGs trigger the same bug, no bug or a bug already known. Column **bugID** refers to IDs in Table 5. The median value is calculated based on the 9 KGs where the minimization was successful.

bugID	#triples input	#triples minimized	#triples removed	share removed	time(s)
P2	4,268	51	4,217	98.81%	63
P4	2,699	170	2,529	93.70%	152
P5	918	12	906	98.69%	4
H1	5,637	2	5,635	99.96%	4
O4 [13]	3,301	54	3,247	98.36%	89
O4 [13]	1,603	92	1,511	94.26%	45
O4 [13]	7,806	30	7,776	99.61%	39
—	2,709	6	2,703	99.78%	3
P4	2,686	404	2,282	84.96%	491
median	2,709	51	2,703	98.47%	45
P1	minimization not possible because bug occurs nondeterminnistic				
P3	memory limit (8GB) exceeded				
—	minimization not possible because anomaly is triggered in parser				

can not be minimized because it triggers an issue in the parser for the KG format and we rely on the same parser for the implementation of the minimization algorithm.

We managed to minimize nine of the ten test inputs for which our minimization algorithm is applicable. Reducing the KG triggering bug P3 required too many resources, i.e., more than 8GB of RAM, which was the memory limit that we used. The reduction of the test inputs took less than 10 minutes for each input with a median time of 45 seconds. In the median, we are able to remove more than 98% of the triples from the inputs (see Table 10).

Table 10 shows how much we were able to reduce the size of the KGs triggering the bugs of Reasoners. In all cases where the minimization was successfull, we are able to remove several hundred or even several thousand triples from the KGs. Although the test-case minimizer does not guarantee to find the overall minimal KG, we still achieve huge reductions. For all cases, the share of triples that got removed exceeds 80%, in most cases it even exceeds 98%. This shows that this method can be very helpful to bring the number of triples to a manageable size.

Answer to RQ5: Minimizing the test inputs is possible in most cases. If the test cases can be minimized, large proportions of the KG can be removed, which makes identifying root causes easier.

6.3.6 RQ6: Can an iterative process be used to develop a robustness mask that characterizes the interaction between the software and the KG?

The robustness masks are able to restrict the allowed test cases. Figure 13 shows that the more the mask is refined, the more attempts need to be made to get a valid mutant. This is to be expected and demonstrates that iterative refinement of a mask really leads

to an increasingly tailored mask while the generated mask is still permissive enough that test cases can pass it with reasonable probability. An interesting observation is that the development of a mask converges with different speeds for different scenarios. While we were able to derive the final mask for Geo after the (initial) batch of 100 test runs, we needed seven iterations of masks and 180 test runs for Suave.

However, we also discovered shortcomings of our procedure. The main problem is that masks expressed using SHACL shapes are not precise enough to characterize all kinds of restrictions. The masks are very good at characterizing parts of the KG that should not change but they are less suited to describe more complex restrictions. Some mutations might be fine in isolation but can cause problems in combination. For example, the robot in Suave has several strategies to find the pipeline, which are represented in the KG. Each of them can be deleted from the KG and the robot can still find the pipeline. But if all strategies are deleted, this is no longer the case.

One difficulty for the iterative process is to find the best number of mutation operators to apply to the initial KG. The higher the number, the more test runs fail and we can get insights into the behavior of the SUT faster. Therefore, less test runs of the system are required. But on the other hand, it is harder to identify which of the mutations led to the failure. We identified such mutations by finding mutations in failing runs that do not occur in passing runs. As mentioned in the previous paragraph, there might be dependencies between different mutations, which further complicates this task. Therefore, one wants to use as many mutation operators as possible while only having as many that one can still identify the ones that cause a failure. For systems where one does not care about which mutation caused the issue, such as Reasoners, a large number of mutation operators is less of a problem, but still increases the time to generate a mutant, and thus the number of tests that can be performed in a given time. We identified that for the systems Suave and Geo the optimal number of applied mutation operators happened to be two and for Reasoners we chose 30 as the confidence intervals of our tests (see Table 7) suggest that we cover more than 90% of the features with 100 mutants using this number.

Answer to RQ6: The iterative process can derive masks that are able to characterize significant restrictions on the SUT. On the contrary, there are some restrictions to what can be expressed with the masks and the optimal number of mutations to generate a mutant is not obvious.

6.3.7 RQ7: To what extent can mutation operators be extracted from reference KGs?

There are two dimensions to evaluate to what extent extracted mutation operators can be used: (i) the quality of the extracted operators and (ii) how well the extraction process works.

We evaluate the quality of the extracted mutation operators using Reasoners. Overall, the quality of the extracted operators is not as good as the quality of the operators designed by domain experts. Nevertheless, the extracted operators can provide new insights into the behavior of the SUTs. The extracted operators lead to more test inputs that are invalid, i.e., do not represent EL ontologies. About 34% of test inputs are not valid, compared to 11% when using operators designed by domain experts.

$$(?b\ \texttt{rdf:type}\ \texttt{rdf:List}) \wedge (?a\ \texttt{rdf:rest}\ ?b) \rightarrow (?a\ \texttt{rdf:type}\ \texttt{rdf:List}) \quad (1)$$

$$(\texttt{tomasys:o\_status}\ \texttt{rdf:type}\ ?b) \wedge (?a\ \texttt{rdf:type}\ \texttt{owl:DatatypeProperty}) \rightarrow (?a\ \texttt{rdf:type}\ ?b) \quad (2)$$

Fig. 16: Example association rules mined from Suave KGs

Furthermore, KGs generated using the extracted operators cover fewer input features (see Section 6.3.4). Nevertheless, we were able to trigger three of the bugs using the extracted operators (see Section 6.3.2). We also found that the extracted operators can produce KG structures that can not be generated using the hand-crafted operators. Namely, object-property chains that contain only one element can only be generated using the extracted operators. This shows that extracting operators can lead to mutation operators that are overlooked by domain experts.

The performance of the extraction process depends on the amount of reference KGs that are available. Extracting mutation operators for Reasoners from the 307 reference KGs took about 6 minutes. Also, finding suitable parameters was much faster, i.e., requiring about 15 minutes, than manually designing mutation operators. However, we failed to extract mutation operators for Suave. The problem was extracting meaningful association rules. Because the KG for Suave is fairly small, the extracted association rules were either too general and did not represent a domain-specific relation (see rule (1) in Figure 16) or were not meaningful at all. An example of the latter is rule (2) in Figure 16, where the only possible matching for variable `?b` is `owl:DatatypeProperty` because that is the only type of the individual `tomasys:o_status`. Thus, this rule means that every node (`?a`) that is of type `owl:DatatypeProperty` is also of type `owl:DatatypeProperty`, which is meaningless. The poor quality of the association rules is caused by the fact that the Suave KG is small and the domain-specific patterns are not repeating enough to be extracted.

Answer to RQ7: Mutation operators that are of practical value can be automatically extracted if enough reference KGs are provided. This reduces human effort but leads to mutation operators with lower input feature coverage.

6.4 Threats to Validity

While we made efforts to set up the experiments in a way that allows for general conclusions, we are aware of some potential threats to the validity of our results. In the following, we consider both threats to the *internal* and *external* validity of the reported results.

Internal Validity

Some threats to internal validity stem from the fact that our testing method relies on human knowledge about the SUTs to configure the testing architecture. This involves

(i) choosing the test profile, (ii) the refinement steps of the mask and (iii) determining when to stop refining the mask.

i The test profiles, in particular the set of mutation operators, is designed based on knowledge about the SUTs. Using different operators might have revealed different or more insights into the behavior of the systems than the insights we gathered in our evaluation.
ii The process of developing a robustness mask requires a human with insight into the semantics of the KG. Hence, other users with different backgrounds might develop different masks and gather different insights about the SUTs, i.e. their findings might not be identical with ours.
iii Deciding if a mask really captures the behavior of the software is a difficult task. We decided to stop the refinement of the mask after we did not get any negative oracles out of a batch run of the SUT. However, it might be the case that one would still be able to find a test input for which the SUT does not behave as expect if one uses a larger batch of test cases.

The setup of the testing campaign of Reasoners was influenced by our previous work on testing OWL-EL reasoners [13]. This impacted our choice of which reasoners to test, which OWL profile to consider and what kind of oracle was deemed suitable. Those choices might have prevented us from finding even more bugs in the OWL reasoners.

Similarly, the performance of the automatic extraction is evaluated only on three case studies. This is partly mitigated as these case studies come from very different domains and use cases (robotics and ROS, geology and simulation, domain-less OWL ontologies), but generalization may require further studies.

External Validity

It is not guaranteed that our results hold for all software that makes use of KGs.

We selected systems for our evaluation that we were already familiar with and that we see as typical representatives for software interacting with KGs, i.e., our selection is not a random sample of systems. It is thus possible that our developed testing method works better on those systems than in general; i.e., the method might be more tailored to the kinds of systems that we see as typical for systems that interact with KGs.

The three systems that we used are from very different domains but share some similarities. For example, all systems contain RDF triples representing OWL axioms for schema information but other KGs might use a different encoding of the schema information. Systems that use very different semantics for the triples in the KG might be less suited to be tested with our method.

The systems that we investigated use complementary methods to access the KGs: Suave translates the nodes into Python classes and objects, Geo uses the contained OWL axioms to reflect on its program state and Reasoners uses an API to interpret the KG purely as an OWL ontology. Other software might use yet another method, which might impact how well it can be analyzed by our method.

The mutation operators that we used cover a broad spectrum but we could not use all possible kinds of operators for our systems. In particular, operators that are domain-specific and target the TBox are missing. The effect of our method on systems

where such operators are required is therefore unknown but given that all other kinds of operators have proved helpful, we do not expect particular problems using such operators.

We evaluated the test-case minimizer on several inputs that trigger bugs. The minimizer works well on these 12 KGs and the minimization algorithm has a low theoretical complexity. However, it is not guaranteed that similar reductions can be achieved for all kinds of KGs due to the small sample size.

7 Conclusion

We presented an approach for integration testing of software that interacts with KGs. We consider integration testing and combine mutation operators for KGs with robustness masks to generate meaningful mutants. We are able to define the robustness mask and test two highly complex systems, including one simulated robotic system.

For applications that operate on general KGs, we can use the mutation operators to find bugs in mature software, namely EL-reasoners. For this class of software, we were able to find additional bugs, compared to prior work using language-based testing [13]. Our evaluation showed that the combination of mutation operators and robustness masks allows to identify and describe KGs for which the SUT does not behave as expected.

On the conceptual level, we have used the number of used terms of the input ontology as a measure for input coverage, and have also introduced a test-case minimizer that enables to reduce test cases to determine the root cause of a bug.

Both approaches investigating for software operating on KGs, i.e., the mutation-based one presented here and the language-based one [13], are black-box approaches that do not consider code coverage. For future work, white-box and black-box approaches remain to be investigated.

Declarations

Funding

This work was partially supported by the SM4RTENANCE EU project (101123490) and by the project REMARO that has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No 956200.

Author Contributions

Tobias John: Writing – review & editing, Writing – original draft, Conceptualization, Methodology, Software, Investigation, Data Curation, Formal analysis, Visualization. *Einar Broch Johnsen:* Writing – review & editing, Writing – original draft, Funding acquisition, Conceptualization, Methodology. *Eduard Kamburjan:* Writing – review & editing, Writing – original draft, Conceptualization, Methodology, Software.

Data Availability Statement

The implementation and data for reproducing our results are available on Zenodo [72]. The source code is also available on Github (https://github.com/smolang/RDFMutate/tree/ese).

Conflicts of interests/Competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Ethical Approval

not applicable

Informed Consent

not applicable

Clinical Trial Number

not applicable

References

[1] Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., 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 Computing Surveys **54**(4), 71–17137 (2022) https://doi.org/10.1145/3447772

[2] Hitzler, P.: A review of the semantic web field. Communications of the ACM **64**(2), 76–83 (2021) https://doi.org/10.1145/3397512

[3] Gangemi, A.: Ontology design patterns for semantic web content. In: The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, Proceedings. Lecture Notes in Computer Science, vol. 3729, pp. 262–276. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11574620_21

[4] Corman, J., Reutter, J.L., Savkovic, O.: Semantics and validation of recursive SHACL. In: ISWC (1). Lecture Notes in Computer Science, vol. 11136, pp. 318–336. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00671-6_19

[5] Silva, G.R., Päßler, J., Zwanepol, J., Alberts, E., Tarifa, S.L.T., Gerostathopoulos, I., Johnsen, E.B., Corbato, C.H.: SUAVE: An exemplar for self-adaptive underwater vehicles. In: 2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 181–187 (2023). https://doi.org/10.1109/SEAMS59076.2023.00031

[6] Qu, Y., Kamburjan, E., Torabi, A., Giese, M.: Semantically triggered qualitative simulation of a geological process. Applied Computing and Geosciences **21**, 100152 (2024) https://doi.org/10.1016/j.acags.2023.100152

[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] 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

[9] 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

[10] Shashank, S.P., Chakka, P., Kumar, D.V.: A systematic literature survey of integration testing in component-based software engineering. In: 2010 International Conference on Computer and Communication Technology (ICCCT), pp. 562–568 (2010). https://doi.org/10.1109/ICCCT.2010.5640467

[11] John, T., Johnsen, E.B., Kamburjan, E.: Mutation-based integration testing of knowledge graph applications. In: 35th IEEE International Symposium on Software Reliability Engineering, ISSRE 2024, pp. 475–486. IEEE, New York City, NY, USA (2024). https://doi.org/10.1109/ISSRE62328.2024.00052

[12] Zeman, V., Kliegr, T., Svátek, V.: RDFRules: Making RDF rule mining easier and even more efficient. Semantic Web **12**(4), 569–602 (2021) https://doi.org/10.3233/SW-200413

[13] John, T., Johnsen, E.B., Kamburjan, E., Steinhöfel, D.: Language-based testing for knowledge graphs. In: 22nd European Semantic Web Conference (ESWC 2025). Lecture Notes in Computer Science, vol. 15719, pp. 24–46. Springer, Berlin, Heidelberg (2025). https://doi.org/10.1007/978-3-031-94578-6_2

[14] Baader, F., Horrocks, I., Lutz, C., Sattler, U.: An Introduction to Description Logic. Cambridge University Press, Cambridge (2017). https://doi.org/10.1017/9781139025355

[15] Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, Cambridge, England (2003). https://doi.org/10.1017/CBO9780511711787

[16] Lonetti, F., Marchetti, E.: Chapter Three - Emerging Software Testing Technologies. Advances in Computers **108**, 91–143 (2018) https://doi.org/10.1016/BS.ADCOM.2017.11.003

[17] Beizer, B.: Software System Testing and Quality Assurance. Van Nostrand Reinhold Co., USA (1984)

[18] Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys **29**(4), 366–427 (1997) https://doi.org/10.1145/267580.267590

[19] Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. John Wiley & Sons, Hoboken, New Jersey, USA (2011). https://doi.org/10.1002/9781119202486

[20] Kuhn, R., Kacker, R.N., Lei, Y., Simos, D.E.: Input space coverage matters. Computer **53**(1), 37–44 (2020) https://doi.org/10.1109/MC.2019.2951980

[21] Havrikov, N., Zeller, A.: Systematically covering input structure. In: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, pp. 189–199. IEEE, New York City, NY, USA (2019). https://doi.org/10.1109/ASE.2019.00027

[22] Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: The Fuzzing Book. CISPA Helmholtz Center for Information Security, Saarbrücken, Germany (2024). Retrieved 2024-01-18. https://www.fuzzingbook.org/ Accessed 2024-01-18

[23] ISO, I.: IEEE, systems and software engineering–vocabulary. ISO/IEC/IEEE 24765:2017(E), 1–541 (2017) https://doi.org/10.1109/IEEESTD.2017.8016712

[24] Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering **41**(5), 507–525 (2015) https://doi.org/10.1109/TSE.2014.2372785

[25] Chen, T.Y., Kuo, F., Liu, H., Poon, P., Towey, D., Tse, T.H., Zhou, Z.Q.: Metamorphic testing: A review of challenges and opportunities. ACM Computing Surveys **51**(1), 4–1427 (2018) https://doi.org/10.1145/3143561

[26] Slutz, D.R.: Massive stochastic testing of SQL. In: VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, pp. 618–622. Morgan Kaufmann, Burlington, Massachusetts, USA (1998). http://www.vldb.org/conf/1998/p618.pdf Accessed 2025-06-26

[27] Keet, C.M., Lawrynowicz, A.: Test-driven development of ontologies. In: The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016, Proceedings. Lecture Notes in Computer Science, vol. 9678, pp. 642–657. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34129-3_39

[28] Davies, K., Keet, C.M., Lawrynowicz, A.: TDDonto2: A test-driven development plugin for arbitrary TBox and ABox axioms. In: The Semantic Web: ESWC

2017 Satellite Events, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10577, pp. 120–125. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70407-4_23

[29] Bartolini, C.: Mutating OWLs: Semantic mutation testing for ontologies. In: Proceedings of the International Workshop on domAin Specific Model-based AppRoaches to vErificaTion and validaTiOn, AMARETTO@MODELSWARD 2016, pp. 43–53. SciTePress, Setúbal, Portugal (2016). https://doi.org/10.5220/0005844600430053

[30] Porn, A.M., Peres, L.M.: Semantic mutation test to OWL ontologies. In: Proceedings of the 19th International Conference on Enterprise Information Systems, pp. 434–441. SCITEPRESS - Science and Technology Publications, Porto, Portugal (2017). https://doi.org/10.5220/0006335204340441

[31] 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: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 46–14612. ACM, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3623307

[32] Kamm, M., Rigger, M., Zhang, C., Su, Z.: Testing graph database engines via query partitioning. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 140–149. ACM, New York, NY, USA (2023). https://doi.org/10.1145/3597926.3598044

[33] 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: ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 302–313. ACM, New York, NY, USA (2022). https://doi.org/10.1145/3533767.3534409

[34] Zhuang, Z., Li, P., Ma, P., Meng, W., Wang, S.: Testing graph database systems via graph-aware metamorphic relations. Proceedings of the VLDB Endowment **17**(4), 836–848 (2023) https://doi.org/10.14778/3636218.3636236

[35] Hua, Z., Lin, W., Ren, L., Li, Z., Zhang, L., Jiao, W., Xie, T.: GDsmith: Detecting bugs in cypher graph database engines. In: Just, R., Fraser, G. (eds.) Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, pp. 163–174. ACM, New York, NY, USA (2023). https://doi.org/10.1145/3597926.3598046 . https://doi.org/10.1145/3597926.3598046

[36] Yang, R., Zheng, Y., Tang, L., Dou, W., Wang, W., Wei, J.: Randomized differential testing of RDF stores. In: 45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, pp. 136–140. IEEE, New York City, NY, USA (2023). https://doi.org/10.1109/ICSE-COMPANION58688.2023.00041

[37] Liu, S., Lan, J., Du, X., Li, J., Lu, W., Jiang, J., Du, X.: Testing graph database systems with graph-state persistence oracle. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, pp. 666–677. ACM, New York, NY, USA (2024). https://doi.org/10.1145/3650212.3680311

[38] 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: IEEE Conference on Software Testing, Verification and Validation, ICST 2024, pp. 25–36. IEEE, New York City, NY, USA (2024). https://doi.org/10.1109/ICST60714.2024.00012

[39] Mansur, M.N., Wüstholz, V., Christakis, M.: Dependency-aware metamorphic testing of datalog engines. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 236–247. ACM, Seattle, WA, USA (2023). https://doi.org/10.1145/3597926.3598052

[40] 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

[41] Armstrong, T.G., Ponnekanti, V., Borthakur, D., Callaghan, M.: LinkBench: a database benchmark based on the Facebook social graph. In: Ross, K.A., Srivastava, D., Papadias, D. (eds.) International Conference on Management of Data (SIGMOD Conference), pp. 1185–1196. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2463676.2465296 . https://doi.org/10.1145/2463676.2465296

[42] Raynaud, T., Amir, S., Haque, R.: A generic and high-performance RDF instance generator. International Journal of Web Engineering and Technology **11**(2), 133–152 (2016) https://doi.org/10.1504/IJWET.2016.077342

[43] 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, Washington, D.C., USA (2018). https://doi.org/10.1609/AAAI.V32I1.11872

[44] 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, USA (2018). http://proceedings.mlr.press/v80/you18a.html Accessed 2025-06-26

[45] 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. Lecture Notes in Computer Science, vol. 11137, pp. 359–375. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00668-6_22

[46] Samanta, B., De, A., Jana, G., Gómez, V., Chattaraj, P.K., Ganguly, N., Gomez-Rodriguez, M.: NEVAE: A deep generative model for molecular graphs. Journal of Machine Learning Research **21**, 114–111433 (2020). https://jmlr.org/papers/v21/19-671.html. Accessed 2025-06-26

[47] Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H.L., Lemay, A., Advokaat, N.: gMark: Schema-driven generation of graphs and queries. IEEE Transactions on Knowledge and Data Engineering **29**(4), 856–869 (2017) https://doi.org/10.1109/TKDE.2016.2633993

[48] 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$^{\mathrm{X}}$s). IEEE Access **9**, 5609–5639 (2021) https://doi.org/10.1109/ACCESS.2020.3048186

[49] Vecovska, M., Jovanovik, M.: Rdfgraphgen: A synthetic RDF graph generator based on SHACL constraints. Computing Research Repository **abs/2407.17941** (2024) https://doi.org/10.48550/ARXIV.2407.17941 2407.17941

[50] Portisch, J., Paulheim, H.: The DLCC node classification benchmark for analyzing knowledge graph embeddings. In: International Semantic Web Conference (ISWC 2022). Lecture Notes in Computer Science, vol. 13489, pp. 592–609. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19433-7_34

[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). Lecture Notes in Computer Science, vol. 14665, pp. 3–20. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-60635-9_1

[52] Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '08, pp. 206–215. ACM, New York, NY, USA (2008). https://doi.org/10.1145/1375581.1375607

[53] Steinhöfel, D., Zeller, A.: Input invariants. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, pp. 583–594. ACM, New York, NY, USA (2022). https://doi.org/10.1145/3540250.3549139

[54] Lee, S., Bai, X., Chen, Y.: Automatic mutation testing and simulation on OWL-S specified web services. In: 41st Annual Simulation Symposium (ANSS-41 2008), pp. 149–156 (2008). https://doi.org/10.1109/ANSS-41.2008.13

[55] Lemieux, C., Sen, K.: FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 475–485. ACM, New York,

NY, USA (2018). https://doi.org/10.1145/3238147.3238176

[56] Sahoo, A., Dwivedy, S.K., Robi, P.S.: Advancements in the field of autonomous underwater vehicle. Ocean Engineering **181**, 145–160 (2019) https://doi.org/10.1016/j.oceaneng.2019.04.011

[57] Niu, H., Adams, S., Lee, K., Husain, T., Bose, N.: Applications of Autonomous Underwater Vehicles in Offshore Petroleum Industry Environmental Effects Monitoring. Journal of Canadian Petroleum Technology **48**(05), 12–16 (2009) https://doi.org/10.2118/09-05-12-GE

[58] Olivares-Alarcos, A., Beßler, D., Khamis, A., Goncalves, P., Habib, M.K., Bermejo-Alonso, J., Barreto, M., Diab, M., Rosell, J., Quintas, J., Olszewska, J., Nakawala, H., Pignaton, E., Gyrard, A., Borgo, S., Alenyà, G., Beetz, M., Li, H.: A review and comparison of ontology-based approaches to robot autonomy. The Knowledge Engineering Review **34** (2019/ed) https://doi.org/10.1017/S0269888919000237

[59] Beetz, M., Beßler, D., Haidu, A., Pomarlan, M., Bozcuoğlu, A.K., Bartels, G.: Know Rob 2.0 — a 2nd generation knowledge processing framework for cognition-enabled robotic agents. In: 2018 IEEE International Conference on Robotics and Automation (ICRA), pp. 512–519 (2018). https://doi.org/10.1109/ICRA.2018.8460964

[60] Zhai, Z., Martínez Ortega, J.-F., Lucas Martínez, N., Castillejo, P.: A Rule-Based Reasoner for Underwater Robots Using OWL and SWRL. Sensors **18**(10), 3481 (2018) https://doi.org/10.3390/s18103481

[61] Coffelt, J., Kashani, M.M., Wąsowski, A., Kampmann, P.: Belief-based fault recovery for marine robotics. In: Proceedings of the Joint Ontology Workshops 2022 Episode VIII: The Svear Sommar of Ontology. CEUR Workshop Proceedings, vol. 3249. CEUR-WS.org, Aachen, Germany (2022). https://ceur-ws.org/Vol-3249/paper3-RobOntics.pdf Accessed 2025-06-26

[62] Baader, F., Suntisrivaraporn, B.: Debugging SNOMED CT using axiom pinpointing in the description logic EL+. In: Proceedings of the Third International Conference on Knowledge Representation in Medicine. CEUR Workshop Proceedings, vol. 410. CEUR-WS.org, Aachen, Germany (2008). https://ceur-ws.org/Vol-410/Paper01.pdf Accessed 2025-06-26

[63] Horridge, M., Parsia, B., Sattler, U.: Explaining inconsistencies in OWL ontologies. In: Scalable Uncertainty Management, Third International Conference, SUM 2009. Lecture Notes in Computer Science, vol. 5785, pp. 124–137. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04388-8_11

[64] Bentley, J.L., Haken, D., Saxe, J.B.: A general method for solving divide-and-conquer recurrences. SIGACT News **12**(3), 36–44 (1980) https://doi.org/10.1145/

1008861.1008865

[65] Niles, I., Pease, A.: Towards a standard upper ontology. In: Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001. FOIS '01, pp. 2–9. ACM, New York, NY, USA (2001). https://doi.org/10.1145/505168.505170

[66] Arp, R., Smith, B., Spear, A.D.: Building Ontologies with Basic Formal Ontology. The MIT Press, Cambridge, MA, USA (2015). https://doi.org/10.7551/mitpress/9780262527811.001.0001

[67] IEEE Robotics and Automation Society: IEEE Standard Ontologies for Robotics and Automation. IEEE Std 1872-2015, 1–60 (2015) https://doi.org/10.1109/IEEESTD.2015.7084073

[68] Galárraga, L., Teflioudi, C., Hose, K., Suchanek, F.M.: Fast rule mining in ontological knowledge bases with AMIE+. The International Journal on Very Large Data Bases **24**(6), 707–730 (2015) https://doi.org/10.1007/S00778-015-0394-1

[69] Zhao, L., Liu, Y.: Design of semantic web distributed retrieval system based on rule reasoning algorithm. In: 2024 4th Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS), New York City, NY, USA, pp. 192–196 (2024). https://doi.org/10.1109/ACCTCS61748.2024.00041 . IEEE

[70] 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: FUZZING, pp. 47–55. ACM, ??? (2023)

[71] Hadar, I., Zamansky, A., Berry, D.M.: The inconsistency between theory and practice in managing inconsistency in requirements engineering. Empir. Softw. Eng. **24**(6), 3972–4005 (2019)

[72] John, T., Johnsen, E.B., Kamburjan, E.: EMSE 2025 Knowledge Graph Mutation Virtual Machine. Zenodo (2025). https://doi.org/10.5281/zenodo.17206524

[73] Macenski, S., Foote, T., Gerkey, B., Lalancette, C., Woodall, W.: Robot operating system 2: Design, architecture, and uses in the wild. Science Robotics **7**(66), 6074 (2022) https://doi.org/10.1126/scirobotics.abm6074

[74] Corbato, C.H.: Model-based self-awareness patterns for autonomy. PhD thesis, Universidad Politécnica de Madrid (2013). https://doi.org/10.20868/UPM.thesis.23178

[75] Sattler, U., Schneider, T., Zakharyaschev, M.: Which kind of module should I extract? In: Proceedings of the 22nd International Workshop on Description Logics (DL 2009). CEUR Workshop Proceedings, vol. 477. CEUR-WS.org, Aachen, Germany (2009). https://ceur-ws.org/Vol-477/paper_33.pdf Accessed

[76] Garcia, L.F., Abel, M., Perrin, M., Dos Santos Alvarenga, R.: The GeoCore ontology: A core ontology for general use in Geology. Computers & Geosciences **135**, 104387 (2020) https://doi.org/10.1016/j.cageo.2019.104387

[77] Horrocks, I., Wu, Z., Grau, B.C., Fokoue, A., Motik, B.: OWL 2 web ontology language profiles (second edition). W3C recommendation, W3C (December 2012). https://www.w3.org/TR/2012/REC-owl2-profiles-20121211 Accessed 2025-06-26

[78] 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

[79] Patel-Schneider, P., Motik, B.: OWL 2 web ontology language mapping to RDF graphs (second edition). W3C recommendation, W3C (2012). https://www.w3.org/TR/2012/REC-owl2-mapping-to-rdf-20121211/ Accessed 2025-06-26

[80] Thompson, H., Malhotra, A., Sperberg-McQueen, M., Peterson, D., Gao, S., Biron, P.V.: W3C XML schema definition language (XSD) 1.1 part 2: Datatypes. W3C recommendation, W3C (April 2012). https://www.w3.org/TR/2012/REC-xmlschema11-2-20120405 Accessed 2025-06-26