Mutation-Based Integration Testing of Knowledge Graph Applications

Tobias John
University of Oslo
Oslo, Norway
tobiajoh@ifi.uio.no

Einar Broch Johnsen
University of Oslo
Oslo, Norway
einarj@ifi.uio.no

Eduard Kamburjan
University of Oslo
Oslo, Norway
eduard@ifi.uio.no

***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 singular 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 employ *domain-specific mutation operators*, that manipulate subgraphs in a domain-adherent way. Second, we need to specify those parts of the knowledge that the software relies on for correctness. We introduce the notion of a *robustness mask* as shapes in the graph that the mutant must conform to. We evaluate our approach on two software applications from the robotic and simulation domain that tightly integrate with their respective KG.**

I. Introduction

a) Motivation: Artificial intelligence (AI) is becoming ubiquitous in modern applications, yet poses challenges for testing the integration of AI components into software. Not only are these systems 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 it. Knowledge graphs (KG) [14], 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. Indeed, their integration with software tools is their next big challenge [13].

b) Contributions: We present a new approach to test the robustness of KG-components integrated with software, in the sense of *integration testing* [31] to investigate whether the software works correctly with unexpected KGs. Hence, our system under test (SUT) is the combination the two interacting components software and KG. Based on a seed KG, we use mutation testing for test-case generation, but retain the original software and test oracle. Thus, we test the integration by fuzzing one component. To address the specific challenges, described below, we (a) introduce the notion of *domain-specific mutators* to mirror possible changes in the KG, and (b) introduce the notion of a *robustness mask* to specify the structural assumptions of the software about the KG and its ontology. The overall goal is to characterize the graph structures that the software component assumes in a KG in order to work correctly.

c) Challenges: Testing the SUTs poses several challenges, due to how the software components interacts with the 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 through 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 not manipulated in isolation, but according to specific patterns [9] and in conjunction with other triples.
3) KGs are *logically embedded*. Not every knowledge graph is also valid – it must 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.

Generation-based fuzzing, e.g., grammar-based fuzzing [11], 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 [33], but these techniques are still not expressive enough to express arbitrary graph structures. Existing mutation testing approaches for the underlying ontology [5], [19], [26] provide an alternative

Tobias John is part of the project REMARO that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 956200.
Eduard Kamburjan was partly funded by the EU project SM4RTENANCE (grant no. 101123423

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.

Facing these challenges, the main research question that we aim to answer in this work is the following:

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

As an answer, we propose a novel architecture for a *test case generator* that is tailored for KGs.

d) Approach: Our generator is based on two key components: (i) a set of mutation operators and (ii) a robustness mask. The first is used to generate random test cases and the second one is used to filter the generated test cases to the meaningful ones.

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 Fig. 1 in Section III.

The different kinds of mutation operators have slightly 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 the generator 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 [7] 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. By using a reasoner to decide this consistency, we address the third challenge.

Based on this approach, we aim to answer the following two research questions in our work:

- **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?

We evaluate our approach on two case studies. First, the SUAVE system for autonomous underwater robots [32] 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 [27], where we are able to capture assumptions on the ontology that specifies the interface between software and knowledge graph.

II. Background

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

A. Preliminaries

a) Knowledge Graphs and Ontologies: Knowledge graphs [14] are a technique for knowledge representation that connects graph *data* with ontological *axioms* (the so-called ontology) [3]. 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 [2] 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.

b) 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) [21]. Here, we focus on integration testing [31], where a test case consists of a *test input* and a *test oracle* that determines whether the combination of at least two components fails or passes the test.

To create test inputs, one can either hand-curate new test input manually, or generate them automatically, a process called *fuzzing* [34]. 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 "the degree to which a system or component can function correctly in the presence of invalid

[1]Resource Description Framework, `www.w3c.org/RDF/`
[2]Web Ontology Language, `www.w3c.org/OWL/`
[3]SPARQL Query Language for RDF, `www.w3c.org/SPARQL`
[4]Shapes Constraint Language, `www.w3.org/TR/shacl/`

inputs or stressful environmental conditions" [15], but in highly constrained situations, mutation-based fuzzing can be considered better suited to avoid generating too many invalid inputs [34].

Designing oracles is a challenging task [4], and if generating the expected output is not possible, one can use indirect techniques, such as metamorphic testing [6], which relates the outputs of different runs to each other, for example the output of an original test case to a generated one.

B. 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 [8], [18]. This procedure is fully independent from the software components eventually using the ontology. Similarly, mutations of ontologies, mostly focusing on the TBox axioms [5], [26], 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 [16], [17]. 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. Metamorphic testing of Datalog engines, while not directly addressing KGs, similarly focuses on generating new instances for software components that work on generic inputs [23].

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 [20]. 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.

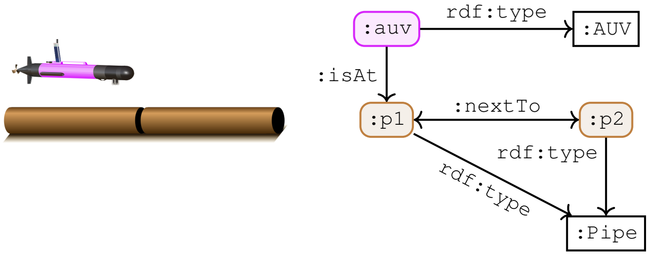
In the next section, we illustrate by a concrete example how this gap manifests itself for tightly coupled systems – software components that rely on the specifics of the used KGs and interact with them in a fixed vocabulary.

III. Motivating Example

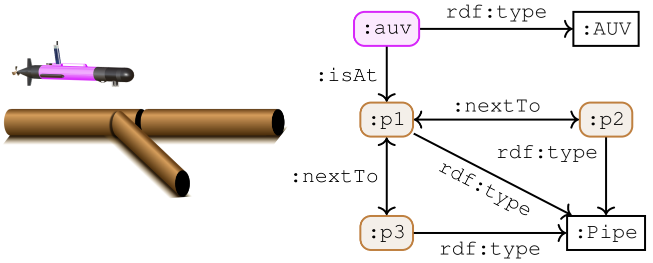
To illustrate tightly coupled and KG-based software applications, their challenges and to give an overview over our approach, consider the software for an autonomous underwater vehicle (AUV), which performs infrastructure inspection in an unknown environment [25], [29]. This is a simplified

```
p := **query**(":isAt(:auv, ?p)")
inspect(p)
S := **query**(":nextTo(p, ?s)")
**while** S $\neq \emptyset$ **do**
    p := S.pop()
    **if** $\neg$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**.



(b) Scenario with corresponding KG representation before mutating.



(c) Scenario with corresponding KG representation after mutating.

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

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

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

system based on the deployed SUAVE system, which we will use in our evaluation in Section VI. 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 [**?**], [**?**], [**?**], [**?**].

a) Infrastructure Inspection: For pipeline inspection, a scenario is depicted in Fig. 1. In Fig. 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 Fig. 1b 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`).

b) Challenges: Testing the robustness of the integration of the software component implementing the algorithm with the KG involves the challenges mentioned in Section I.

(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 Fig. 1c 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 Fig. 1c, which reveals an assumption of the software, differs in several ways from the original KG in Fig. 1b. 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.

c) 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 Fig. 1b and then use mutation operators that are specifically designed for the underwater domain.

We apply *domain-specific mutation operators* to the original KG (Fig 1b) 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 Fig. 1c. So, we can define an operator that generalizes the change from the KG in Fig. 1b to the KG in Fig. 1c. 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 Fig. 1d. 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 Fig. 1c, 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 V-B.

IV. Testing Architecture

We propose to use the *testing architecture*, which is shown in Fig. 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.

A. Structures

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

a) 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.

b) 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 III. 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.*

c) 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 III. We use the graph pattern* $\{(x, \mathrm{type}, \mathrm{Pipe}), (y, \mathrm{type}, \mathrm{Pipe})\}$, *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* $\{(\mathrm{p1}, \mathrm{type}, \mathrm{Pipe}), (\mathrm{p2}, \mathrm{type}, \mathrm{Pipe})\}$, *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

```
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.

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.

Example 3. *We consider again the example from the pipeline domain. We define a mutation operator that connects two pipe sections with the patterns* $\mathrm{S} = \{(x, \mathrm{type}, \mathrm{Pipe}), (y, \mathrm{type}, \mathrm{Pipe})\}$ *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.

d) Graph shapes: We use SHACL to define shapes of KGs [7]. 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 ap-*

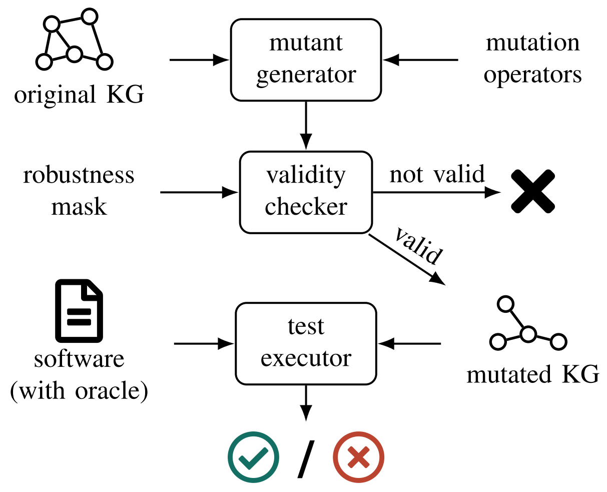


Fig. 3: Testing architecture.

plied 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 Fig. 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.*

B. Components

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

a) Mutant Generator: The *mutant generator* is the central component of our architecture. Its location in the architecture is depicted in Fig. 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.

b) 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 Fig. 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 is checked to be consistent w.r.t. the ontology it contains. An inconsistent KG entails all knowledge and is therefore useless. Because of that, 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.

c) Test Executor: The last component in our architecture (see Fig. 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.

C. 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.

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 [5], [26]. 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 whole space of possible KGs is explored, the domain-independent operators for adding and deleting nodes and relations should be part of the set of operators.

Additionally, the user provides the number of mutations that should be applied to generate the mutant. 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 a 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 consistent mutants are used for testing. More details about how the robustness mask can be obtained are described in Section V-B.

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 does not comply with the oracle needs to be interpreted by the user and is the source of new insights about the interaction, that can be added to the robustness mask.

V. Testing Methodology

To use our testing architecture most effectively, we propose to use it as describe in this section. We discuss to which parts of the KG the mutation operators should be applied to, how to obtain a robustness mask and how to interpret the generated mask.

A. *Hierarchy of Axioms*

Usually, KGs (and in particular the ontologies they contain) are not build from scratch for each application but follow established ways to organize knowledge by importing ontologies that describe more general concepts. This design is also advised in order to allow for easier integration between different KGs. The most general of these ontologies are called *top-level ontologies*, such as SUMO [24] or BFO [1]. 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 [28]. 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.

B. *Obtaining Robustness Masks*

As explained in Section IV, 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.

a) 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 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.

As this iterative approach is an important aspect of using our testing procedure, we aim to evaluate whether it works by answering the following research question:

- **RQ4:** Can the *iterative process* be used to develop a robustness mask that characterizes the interaction between the software and the KG?

b) Example: We demonstrate the iterative approach of developing a robustness mask using our running example from Section III. Remember that the mutant in Fig. 1c 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 Figs. 4a and 4b. 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 Fig. 1c 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 Fig. 5, for which the SUT does not inspect all 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

```
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:node :auv ; ].
```

(d)

Fig. 4: Four SHACL shapes that are useful for the robustness mask of the pipeline inspection example.

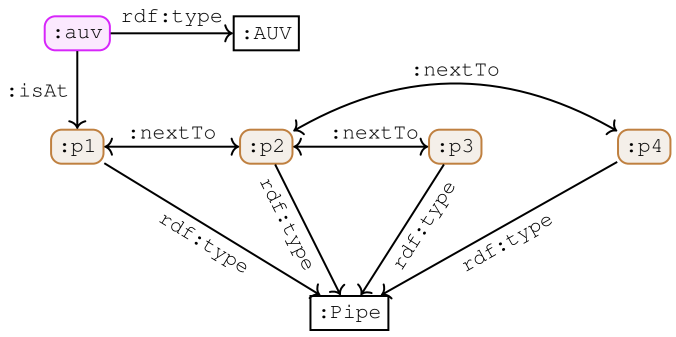


Fig. 5: A mutant of the KG in Fig. 1b with two new pipe segments.

shape in Fig. 4c describes the desired structure. The shape specifies that every pipe segment is connected to at most two nodes via `:nextTo` relations. The mutant in Fig. 5 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 Fig. 4c.

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 Fig. 4d. 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 Fig. 4. Together with the shape that we introduced in Fig. 1d, we finally obtain a robustness mask for our example that describes all requirements of the algorithm on the structure of the KG.

C. *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.

D. *Discussion*

We briefly discuss two further issues: *redundant mutants* and the *human aspect* in our methodology.

Redundant mutants are mutants that reveal the same fault of the SUT. They can be equivalent, but do not have to be. We address the problem of redundant mutants with the robustness masks: After refinement following a failing test run, the mask contains shapes such that this failure is not explored again. The mask not only prevents using the exact same mutant again for testing but defines the equivalence between mutants on a semantic level, i.e. addresses redundant mutants.

Our iterative approach requires the user to refine the mask. Different users might generate different masks when testing the same SUT as there are multiple ways to prevent a specific failure. 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.

VI. Evaluation

A. *Research Questions*

We recall our four 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?

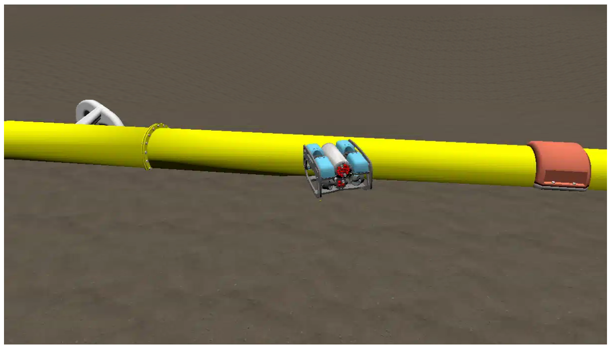


Fig. 6: Underwater robot simulation Suave with an AUV inspecting a pipeline.

- **RQ3:** What are the consequences of using *domain-specific mutation operators* compared to domain-independent mutation operators?
- **RQ4:** Can the *iterative process* be used to develop a robustness mask that characterizes the interaction between the software and the KG?

B. Experimental Design and Setup

We apply our testing methodology to two different systems to answer these research questions. The two 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.

a) SUAVE: The first system is Suave [32]. It is a control system for autonomous underwater robots implemented on top of ROS2 [22]. 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* [12]. 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 architecture [12], 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 [30], 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 Fig. 6. 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.

b) Geo Simulator: The second system that we use for our evaluation is a geological simulator [27]. 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 [1] and the mid-level ontology GeoCore [10]. 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.

c) Mutation Operators: We use 19 mutation operators for our evaluation, which are depicted in Table I. 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 both systems for our evaluation, we 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

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

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
RemoveAxiom	delete an axiom	ABox / TBox	no	Suave
AddRelation	add an object property relation between individuals	ABox	no	Suave
ChangeRelation	change object property relation (change target)	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

TABLE I: Implemented mutation operators. Operators marked with “*” are taken from [26], the other operators are introduced by us.

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 mutation operators used for Geo 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.

d) Implementation: We implemented our framework and the mutation operators using Kotlin. The implementation and data for reproducing our results can be found online [**?**].

For both systems, we generated mutated KGs in batches and updated the robustness mask accordingly after obtaining the results for all the runs in a batch. For Geo we chose a batch size of 100 and for Suave we chose batch sizes between 10–30. The difference comes from a difference in behavior: Test runs using the initial, empty mask failed less often for Geo (29%) than for Suave (50%) and running Geo requires less time (about 5min) than running Suave (about 15min). Therefore, we chose a smaller batch size for Suave to refine the mask faster. Overall, about 15% of the tested mutants lead to failing test runs. For Geo, we generated 10 random scenarios where oil maturation occurs in some but not in all of the scenarios. After obtaining a mask for the first scenario, we also tested it against the other scenarios.

C. Results

For each research question, we state a general answer before providing more evidence for our findings with more detailed insights gained from analyzing Suave and Geo.

a) RQ1: Our presented approach is well suited for integration testing of software with KGs. Using mutations of the original KGs, we identified KGs for which the SUT does not behave as intended for both systems, Suave and Geo, with reasonable effort. The developed masks provide a declarative description of the restrictions, which can be interpreted by KG experts. We could develop such a mask for both systems, Suave and Geo.

b) RQ2: Two types of insight that can be very well detected are parts of the KG that are superfluous and axioms that are missing in the KG.

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.

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.

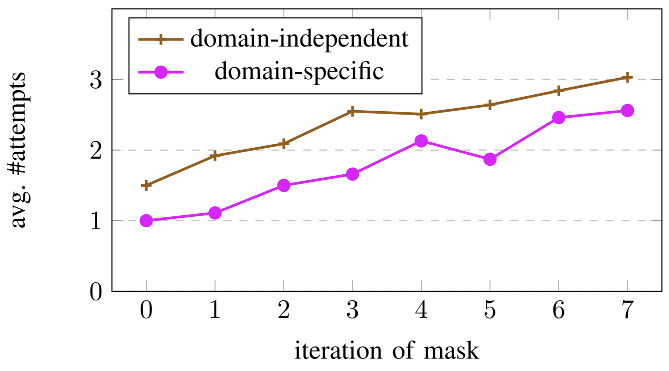


Fig. 7: Average number of attempts to create a valid mutant for each iteration of the mask for Suave. For each data point, batches of 100 mutants were created by applying two domain-specific or domain-independent mutation operators.

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.

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

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 Fig. 7 for Suave. 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 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 RQ4).

d) RQ4: In general, we observed that 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.

The robustness masks are able to restrict the allowed test cases. Fig. 7 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 some 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. We identified that for our systems the optimal number happened to be two.

D. Threats to Validity

It is not guaranteed that our results hold for all kinds of domains where software makes use of KGs but the two systems that we used are from two very different domains. The systems that we investigated use two complementing methods of accessing the KG but other software might use a different method. The mutation operators that we used cover a broad spectrum but we could not use all kinds of possible operators for our systems. In particular, operators that are domain-specific and target the TBox are missing. However, adding mutation operators for this case would only enhance our method, and not invalidate our results. Lastly, the process of developing a robustness mask requires a human with insight into the semantics of the KG. Hence, different users with different backgrounds might develop different masks and gather different insights about the SUTs, i.e. their findings might differ from ours.

VII. Conclusion

We presented an approach for integration testing of software with KGs. The approach combines mutation operators for KGs with a robustness mask to generate meaningful mutants. 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.

We identified several optimizations to make the testing approach more efficient in the future. Using more mutations to generate a mutant increases the probability for a failing run but also makes it harder to extract which mutation caused the failure. Hence, a method of shrinking the set of applied mutations for a failing mutant to the mutations that caused the failure could reduce the compromises to be made in this regard. A second extension could be to use methods of metamorphic testing to avoid the oracle problem. To do so, one would need to relate the mutation operators to the effects that they have on the output of the software.

References

[1] Robert Arp, Barry Smith, and Andrew D. Spear. *Building Ontologies with Basic Formal Ontology*. The MIT Press, July 2015.

[2] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[3] Franz Baader, Ian Horrocks, Carsten Lutz, and Uli Sattler. *An Introduction to Description Logic*. Cambridge University Press, Cambridge, 2017.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.*, 41(5):507–525, 2015.

[5] Cesare Bartolini. Mutating OWLs: Semantic mutation testing for ontologies. In Antonello Calabrò, Francesca Lonetti, and Eda Marchetti, editors, *Proceedings of the International Workshop on domAin specific Model-based AppRoaches to vErificaTion and validaTiOn, AMARETTO@MODELSWARD 2016, Rome, Italy, February 19-21, 2016*, pages 43–53. SciTePress, 2016.

[6] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, 2018.

[7] Julien Corman, Juan L. Reutter, and Ognjen Savkovic. Semantics and validation of recursive SHACL. In *ISWC (1)*, volume 11136 of *Lecture Notes in Computer Science*, pages 318–336. Springer, 2018.

[8] Kieren Davies, C. Maria Keet, and Agnieszka Lawrynowicz. TDDonto2: A test-driven development plugin for arbitrary TBox and ABox axioms. In *ESWC (Satellite Events)*, volume 10577 of *Lecture Notes in Computer Science*, pages 120–125. Springer, 2017.

[9] Aldo Gangemi. Ontology design patterns for semantic web content. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005, Proceedings*, volume 3729 of *Lecture Notes in Computer Science*, pages 262–276. Springer, 2005.

[10] Luan Fonseca Garcia, Mara Abel, Michel Perrin, and Renata Dos Santos Alvarenga. The GeoCore ontology: A core ontology for general use in Geology. *Computers & Geosciences*, 135:104387, February 2020.

[11] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, June 2008. Association for Computing Machinery.

[12] Carlos Hernández. *Model-based Self-awareness Patterns for Autonomy*. Dissertation (external), October 2013.

[13] Pascal Hitzler. A review of the semantic web field. *Commun. ACM*, 64(2):76–83, 2021.

[14] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. *ACM Comput. Surv.*, 54(4):71:1–71:37, 2022.

[15] IEC ISO. IEEE, systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, 2017.

[16] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland H. C. Yap, Zhenkai Liang, and Manuel Rigger. Detecting logic bugs in graph database management systems via injective and surjective graph query transformation. In *ICSE*, pages 46:1–46:12. ACM, 2024.

[17] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. Testing graph database engines via query partitioning. In *ISSTA*, pages 140–149. ACM, 2023.

[18] C. Maria Keet and Agnieszka Lawrynowicz. Test-driven development of ontologies. In *ESWC*, volume 9678 of *Lecture Notes in Computer Science*, pages 642–657. Springer, 2016.

[19] Shufang Lee, Xiaoying Bai, and Yinong Chen. Automatic Mutation Testing and Simulation on OWL-S Specified Web Services. In *41st Annual Simulation Symposium (Anss-41 2008)*, pages 149–156, April 2008.

[20] Caroline Lemieux and Koushik Sen. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *ASE*, pages 475–485. ACM, 2018.

[21] Francesca Lonetti and Eda Marchetti. Chapter Three - Emerging Software Testing Technologies. *Adv. Comput.*, 108:91–143, 2018.

[22] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[23] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. Dependency-Aware Metamorphic Testing of Datalog Engines. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 236–247, Seattle WA USA, July 2023. ACM.

[24] Ian Niles and Adam Pease. Towards a standard upper ontology. In *Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001*, FOIS '01, pages 2–9, New York, NY, USA, October 2001. Association for Computing Machinery.

[25] Haibo Niu, Sara Adams, Kenneth Lee, Tahir Husain, and Neil Bose. Applications of Autonomous Underwater Vehicles in Offshore Petroleum Industry Environmental Effects Monitoring. *Journal of Canadian Petroleum Technology*, 48(05):12–16, May 2009.

[26] Alex Mateus Porn and Leticia Mara Peres. Semantic Mutation Test to OWL Ontologies. In *Proceedings of the 19th International Conference on Enterprise Information Systems*, pages 434–441, Porto, Portugal, 2017. SCITEPRESS - Science and Technology Publications.

[27] Yuanwei Qu, Eduard Kamburjan, Anita Torabi, and Martin Giese. Semantically triggered qualitative simulation of a geological process. *Applied Computing and Geosciences*, 21:100152, March 2024.

[28] IEEE Robotics and Automation Society. IEEE Standard Ontologies for Robotics and Automation. *IEEE Std 1872-2015*, pages 1–60, April 2015.

[29] Avilash Sahoo, Santosha K. Dwivedy, and P. S. Robi. Advancements in the field of autonomous underwater vehicle. *Ocean Engineering*, 181:145–160, June 2019.

[30] Ulrike Sattler, Thomas Schneider, and Michael Zakharyaschev. Which kind of module should I extract? In *Description Logics*, volume 477 of *CEUR*, 2009.

[31] S Phani Shashank, Praneeth Chakka, and D Vijay Kumar. A systematic literature survey of integration testing in component-based software engineering. In *2010 International Conference on Computer and Communication Technology (ICCCT)*, pages 562–568, September 2010.

[32] Gustavo Rezende Silva, Juliane Päßler, Jeroen Zwanepol, Elvin Alberts, S. Lizeth Tapia Tarifa, Ilias Gerostathopoulos, Einar Broch Johnsen, and Carlos Hernández Corbato. SUAVE: An Exemplar for Self-Adaptive Underwater Vehicles. In *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 181–187, May 2023.

[33] Dominic Steinhöfel and Andreas Zeller. Input invariants. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 583–594. ACM, 2022.

[34] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. Retrieved 2024-01-18.