Software Testing Techniques Revisited for OWL Ontologies

Cesare Bartolini(✉)

Interdisciplinary Centre for Security, Reliability and Trust (SnT), Université du Luxembourg, Luxembourg, Luxembourg
cesare.bartolini@uni.lu

Abstract. Ontologies are an essential component of semantic knowledge bases and applications, and nowadays they are used in a plethora of domains. Despite the maturity of ontology languages, support tools and engineering techniques, the testing and validation of ontologies is a field which still lacks consolidated approaches and tools. This paper attempts at partly bridging that gap, taking a first step towards the extension of some traditional software testing techniques to ontologies expressed in a widely-used format. Mutation testing and coverage testing, revisited in the light of the peculiar features of the ontology language and structure, can can assist in designing better test suites to validate them, and overall help in the engineering and refinement of ontologies and software based on them.

Keywords: Mutation testing · Coverage testing · Ontology · OWL · Mutant generation

1 Introduction

The use of semantics in information technology is greatly enhancing the expressiveness of knowledge bases, especially with respect to information representation and retrieval. Information is classified according to domain-specific structures which describe the concepts and the relations between them, and this organization allows an efficient access to such information. Cross-domain organization is also made possible through the use of formal languages to describe the domains. Nowadays, knowledge bases structured according to description logic [1] are popular, and they can also be generated using Natural Language Processing (NLP) techniques to classify unstructured documents.

Semantic knowledge is a wide field of research and application, and it is based on a multi-layered framework of components and technologies. However, at the very basic level, there is the need to describe the domains. This result is achieved by means of *ontologies*. Ontologies are a general concept to denote the definition of a domain, describing it at various level of abstraction.

Of course, to be used in computer systems, ontologies need to be described according to some formal language. Early attempts at defining a language to

S. Hammoudi et al. (Eds): MODELSWARD 2016, CCIS 692, pp. 132–153, 2017.
DOI: 10.1007/978-3-319-66302-9_7

structure knowledge resulted in the Resource Description Framework (RDF) language [2]. However, the purpose of RDF is mainly to describe resources by means of metadata, and it is too low level to provide an efficient means of describing an ontology. For that purpose, the Web Ontology Language (OWL) specification [3] has been defined.

OWL, that was developed starting from another ontology language [4] called DAML+OIL [5], is a family of abstract languages which are expressed in several different syntaxes, some of which are based on eXtensible Markup Language (XML). The primary syntax is RDF/XML, which easily maps onto RDF concepts and integrates with other XML languages.

It is widely known that there is no "right" way of defining an ontology. Its definition really depends on the domain, the desired level of abstraction, the purpose for which the ontology is intended, and a number of choices by the developer. In other words, the same domain could be represented by several totally different ontologies, which would result in different structures of the respective knowledge bases (and consequently, with different results when classifying and querying information). However, for ontology-based applications to be integrated, it is necessary that they are based on the same ontology.

Ontologies have a number of uses, primarily that of describing some domain of knowledge from a specific perspective. In this sense, they act much like a vocabulary, similarly to a database. They have found their place as the basis of knowledge representation in many application fields, from web searches to the medical and legal domains [6].

Ontologies are also used for decision support [7], therefore it is important that they are as complete as possible (within their domain and purpose), and also that they do not contain errors. Previous experiences [8] have highlighted the risks of using an incorrect ontology as a structure for a knowledge base. However, despite the acknowledged importance of the correctness of ontologies, few methodologies and tools exist for the testing and validation of ontologies.

This paper aims at partly filling this void by proposing an extension of some popular software testing techniques to the domain of OWL ontologies. Namely, this work is focused on adapting mutation testing and coverage testing to ontologies.

Mutation testing is a well-known testing method that assesses the validity of a test suite by generating *mutants*, i.e., incorrect versions of the System Under Test (SUT), by introducing single errors in the trustworthy version. The ontology-based software could then be linked to the mutants generated in this way, and run against the test suite. The mutants thus killed can provide important information about the ontology and the program using it, including coverage details and fault detection.

Coverage testing aims at evaluating what portion of the SUT is exercised by a test suite. Such knowledge can be used to determine if the test suite is fit for validating the SUT, or if additional tests need to be designed. On the other hand, coverage testing may reveal some parts of the SUT which are useless or redundant, thus suggesting some possible optimizations. This paper does

not introduce a detailed approach for measuring the coverage of ontologies, but rather a preliminary idea focused mainly on classes, leaving further developments to future work.

The paper is organized as follows. Section 2 provides a survey of existing literature in ontology testing and mutation testing. Section 3 offers a high-level description of mutation testing. Section 3.1 describes the proposed methodology, explaining the various operators used for the mutation of an OWL ontology. It also contains a high-level description of the implementation of the mutation tool. Section 4 proposes a basic methodology to measure the coverage of an ontology by a test suite, with a sample application in Sect. 4.1. Section 5 shows the methodology in action, applying the mutation operators to various ontologies in different domains. Finally, Sect. 6 summarizes the results and envisions some directions for future research.

2 Related Work

Although knowledge bases and semantic applications are a very consolidated domain nowadays, it appears that there has been little attention to the validation of ontologies [9].

The World Wide Web Consortium (W3C) provides a set of test cases for evaluating the OWL ontology from a structural point of vies [10]. [11] defines an algorithm to "debug" ontologies in search of inconsistent classes. [12] offer a means of ontology validation through user-defined test cases, whereas [13] defines an approach to merge large ontologies and find inconsistencies.

A lot of research addresses metrics and benchmarks for ontologies. The work proposed by [14] defines some measures for assessing an ontology, and evaluates these measures by means of a *meta-ontology* against which the ontology under validation is compared. This work does not seem to address the semantic correctness of the ontology but mainly its structure and engineering methodology. A similar approach, but with a greater attention to semantics, is proposed by [15]. [16] defines a benchmark for the analysis of ontologies based on two different semantics, OWL Lite and OWL DL.

In [9], the authors propose a methodology and tool for testing an ontology. The methodology addresses three main perspectives: verification of the Competency Question (CQs) to which the ontology is supposed to provide an answer, verification of the inferences by means of an OWL reasoner, and provocation of errors. The last perspective differs significantly from the current work because it does not modify the ontology structure, but rather introduces test data that are inconsistent with the ontology.

A significant model-checking methodology to validate the design of an ontology is OntoClean [17]. It consists in introducing annotations in the ontology which allow to perform a consistency check.

An interesting approach is described in [18]. The authors have built a testing tool which tries to search for potential pitfalls in ontology development. The list of pitfalls has been introduced by the authors in [19]. Although different from

the idea of introducing errors in the ontology, their work can provide interesting suggestions for the definition of mutation operators.

An approach that combines ontology evaluation with software engineering techniques is described by [20], which introduces a proposal to adapt unit testing to OWL ontologies. In the past, several tools have been developed for ontology unit testing, although it does not appear to be a mainstream testing approach for ontologies. Another interesting approach is presented in [21]: instances are generated from an ontology, and hypotheses are formulates on these instances. The validation of the generated hypotheses is then fed as an input to refine the ontology.

Some previous work concerning mutation testing in the OWL language can be found in [22]. The methodology does not apply to the general ontology language OWL, but rather to a specific ontology called OWL-S [23] which can be used as a semantic descriptor for web services, and it applies mutation to classes, conditions, control flows and data flows. The purpose of that paper is not to improve an ontology and its related test suite, but rather to detect errors in the web service specification. However, some of the concepts introduced in that work are similar to those introduced in the current work.

Concerning coverage testing, again, there does not appear to be any relevant work on the topic. As per mutation testing, some works exist to measure the coverage of web services in OWL-S [24], but the issue of measuring the actual coverage of an ontology appears to be unexplored so far.

3 Mutation Testing

Mutation testing is a testing technique originally proposed in [25,26], although allegedly the initial idea can be traced back to a few years earlier [27]. It is classified either among the syntax-based testing techniques [28], or among the error-based or fault-based testing techniques [29,30]. It is normally, but not exclusively, meant for unit testing [31].

In its essence, it is a methodology in which small parts of a software code are changed. Its main purpose is not to test the SUT proper, but the quality of its test suite. However, it has an indirect benefit on the SUT, because the detection of faults in the test suite can often also lead to detecting errors in the SUT.

According to the description provided by [28], mutation is carried out by applying a set of *mutation operators* to a *ground string*. The ground string is expressed in the grammar, and a mutation operator is "[a] rule that specifies syntactic variations of strings generated from a grammar". These operators can also be applied directly to the grammar if no ground string exists. Mutation can be used to generate both invalid strings and strings that are valid but different from the ground string. In both cases, the strings thus generated are called *mutants*.

The mutants generated from the SUT are then executed on the test suite, and the test results are compared against those of the original code. Those mutants which behave differently with respect to the test suite are *killed* by

the test suite. An ideal test suite would kill n out of n generated mutants. The whole process is generally automated by means of batch scripts, because the generation of a high number of mutants and the execution of the test suite on each is a complex and tedious process which is well-suited for automatization. Mutation can be also carried out by introducing simplifications that reduce the number of mutants [32,33] to lower the complexity of the testing process.

Mutation testing has generally been applied to software code, particularly to Java [34,34]. Previous research [28,35] has identified a set of operators for mutation.

Traditional mutation testing operates at the syntax level, by introducing errors in the code. However, techniques for semantic mutation testing have also been defined [36–38], in which mutation operators affect the semantics of the code. In other words, the code is still syntactically correct, but its functionality is different from the intended one.

3.1 Mutation Testing Applied to OWL

To apply the mutation testing methodology to an ontology, some premises are in order.

First off, the mutation operators will be applied to the ontology. However, the testing can be carried out in two different ways: either by viewing the ontology as the SUT, independently of what it is used for; or when the SUT is the knowledge base or software that relies upon the ontology. Choosing either perspective has significant consequences in the testing and the test suite that is used.

The mutation proposed in this paper is a kind of semantic mutation. The syntax of an ontology is managed satisfactorily by the various parsers and editors available, so unless the SUT is a new OWL editor or parser there would be little need for a syntactic mutation testing. What is significantly more interesting is the evaluation of the ontology definition. Additionally, using OWL as the underlying specification, there is no point in working at the syntax level because OWL does not have a syntax *per se*, but can be built according to different syntaxes. In fact, the proposed methodology has been executed using the OWL/RDF, OWL/XML and Manchester [39] syntaxes with identical results.

The mutation operators have therefore been defined as a set of operations that conceptually modify the ontology. An ontology refers to *entities*, which are the main building blocks used to represent real-world objects. The ontology does not define the entities, which are defined by the domain itself. For the purposes of this work, the following entity types have been used as the ground string for mutation:

classes represent the core concepts in the ontology. A class is the abstraction which subsumes all individuals of a given type;
individuals are the real-world objects, single instances of a class;
object properties describe the relationships between individuals;
data properties are used to associate information data to classes.

In addition to entity-specific mutation operators, it is also possible to define some general operators. In particular, some static information can be added to any entity by means of *annotations*. Typical annotations include *label* and *comment*, which are part of RDF Schema (RDFS) and are language specific.

All mutation operators affect some *axiom*, which is the base expression in the ontology. Axioms are connections between entities, and some examples of axioms are:

- a *subclass* relationship between two classes;
- the belonging of an individual to a class;
- the *domain* or the *range* of an object or data property;
- association of an annotation with its entity.

3.2 Mutation Operators

This section describes the various classes of mutation operators defined for OWL mutation testing. Entities in OWL can be declared using either a human-readable Internationalized Resource Identifier (IRI), or an auto-generated one. When using the latter naming convention, which is recommended by the Protégé software, the domain-specific names must be referred to by means of label annotations. This solution is very versatile, because it does not force a naming, but an entity can have a number of names, also in different languages. However, when referring to entities using labels, the absence of a label can cause errors.

Table 1 offers an overview of all the mutation operators.

Some of the mutation operators produce identical mutants: for instance, the ORI operator, when applied to a class and to its inverse, generates two identical mutants.

Entities. Some mutation operators are general and can be applied to any entity:

ERE. Remove entity. This operator deletes the declaration of an entity from the ontology, be it a class, property, or individual. All axioms concerning the deleted entity are removed as well.

ERL. Remove label. This operator removes a label annotation from an entity.

ECL. Change label language. A label annotation is composed by the actual label and a language attribute. This operator removes the language attribute, setting it to a meaningless value.

While it is possible to also apply mutation operators to comment annotations, comments are generally not meant for processing purposes, but only to provide a description to the human user. Therefore, no mutation on comment annotations has been introduced in this work. Similarly, no mutation operators have been defined for other annotations such as *versionInfo* or *seeAlso*.

Table 1. List of mutation operators.

Entity	Operator	Effect
Any entity	ERE	Remove the entity and all its axioms
	ERL	Remove entity labels
	ECL	Change label language
Class	CAS	Add a single subclass axiom
	CRS	Remove a single subclass axiom
	CSC	Swap the class with its superclass
	CAD	Add disjoint class
	CRD	Remove disjoint class
	CAE	Add equivalent class
	CRE	Remove equivalent class
Object property	OND	Remove a property domain
	ONR	Remove a property range
	ODR	Change property domain to range
	ORD	Change property range to domain
	ODP	Assign domain to superclass
	ODC	Assign domain to subclass
	ORP	Assign range to superclass
	ORC	Assign range to subclass
	ORI	Remove inverse property
Data property	DAP	Assign property to superclass
	DAC	Assign property to subclass
	DRT	Remove data type
Individual	IAP	Assign to superclasses
	IAC	Assign to subclasses
	IRT	Remove data type

Classes. Classes are entities which describe the conceptual abstraction of real-world objects. Class relations can be described in hierarchical terms, from the general to the particular. In other words, a class can be defined as the subclass of another class, by means of an "is a" relationship. Classes can be subclasses of more than one superclass. If a class is not defined as a subclass, then it is implicitly a subclass of the top-level class, *Thing*. A class can also be the subclass of an anonymous class, i.e., a class defined "on the fly" using properties.

In addition to the mutation operators applicable to all entities, the following operators have been defined for class entities:

CAS. Add subclass axiom. This operator introduces a subclass axiom between one class and any other class of which it is not already asserted as being a subclass.

CRS. Remove subclass axiom. This operator removes a subclass axiom, thus changing the hierarchical structure of the ontology. If the class has a single superclass, then it will become a subclass of the top-level class.
CSC. Swap subclass axiom. This operator exchanges a class with one of its superclasses. Simply put, it reverses part of the hierarchical structure.
CAD. Add disjoint class. A class can be asserted as being disjoint with other classes. This operator introduces a disjointness relation between one class and another with which the former is not already disjoint.
CRD. Remove disjoint class. This operator erases a disjoint declaration, so the two classes are no longer disjoint.
CAE. Add equivalent class. A class can be asserted as being equivalent to other classes. This operator introduces an equivalence relation between one class and another to which the former is not already equivalent.
CRE. Remove equivalent class. This operator erases an equivalent declaration, so the two classes are no longer equivalent.

Object Properties. Object properties represent relations between classes which cannot be in hierarchical terms. All relations except "is a" must be defined in terms of object properties.

An object property normally has at least one *domain* and one *range*. The domain represents the classes (which can also be anonymous classes, defined for example using set operations) to which the object property applies. A range represents the possible values that the property can have. In other words, domain and ranges are limitations to the individuals to which the property can be applied and to the individuals that it can have as its values, respectively.

The following mutation operators specific to object properties have been defined:

OND. Remove domain. One domain (set of entities to which the property can apply) is removed from the object property. Since the actual domain is the intersection of all ranges, this operator actually widens the possible entities to which the property can apply.
ONR. Remove range. One range is removed from the object property. Since the actual range is the intersection of all ranges, this operator actually widens the possible values that the property can have.
ODR. Change domain to range. One of the domains of the property is changed to a range, actually restricting its possible values but increasing the classes it can apply to.
ORD. Change range to domain. One of the ranges of the property is changed to a domain.
ODP. Assign to superclass. One of the domains of the property is replaced with one of the superclasses of that domain. This operator cannot be applied to anonymous domains or to domains which are only subclass of the top-level class.
ODC. Assign to subclass. One of the domains of the property is replaced with one of the subclasses of that domain. This operator cannot be applied to anonymous domains.

ORP. Set range to superclass. One of the ranges of the property is replaced with one of the superclasses of that range. This operator cannot be applied to anonymous ranges or to range which are only subclass of the top-level class.

ORC. Set range to subclass. One of the ranges of the property is replaced with one of the subclasses of that range. This operator cannot be applied to anonymous ranges.

ORI. Remove inverse property. The property can be declared as being inverse to another one. This operator removes the inverse declaration, but it does not remove the other property.

Data Properties. Data properties are used to describe additional features of an entity. Technically, they represent a connection between entities and literals (such as XML strings and integers). Data properties have a *domain* which limits the entities it can be applied to, and a range which limits the set of possible literals it can have as values.

In addition to the general operators, the following operators have been defined for data properties:

DAP. Assign to superclass. One of the domains of the property is replaced with one of the superclasses of that domain. This operator cannot be applied to anonymous domains or to domains which are only subclass of the top-level class.

DAC. Assign to subclass. One of the domains of the property is replaced with one of the subclasses of that domain. This operator cannot be applied to anonymous domains.

DRT. Remove data range. One of the data ranges of the property is removed, and it is implicitly replaced with the top-level literal *rdfs:Literal*, actually increasing the set of possible literals that this property can have.

Individuals. Individuals represent single instances of a class (including anonymous classes). Individuals are very similar to classes, but they represent a single object and not an abstract generalization. Therefore, they can be defined as belonging to one or more classes.

The following specific operators have been defined for individuals.

IAP. Assign to superclass. One of the types of the individual is replaced with one of its superclasses. This operators can be applied only to those types which have a superclass different from the top-level class.

IAC. Assign to subclass. One of the types of the individual is replaced with one of its subclasses.

IRT. Remove type. One of the types to which the individual belongs is removed (both named and anonymous classes). If the individual is of a single type, then it becomes an individual of the top-level class.

4 Measuring the Coverage

Although not strictly related to mutation testing, coverage testing [40,41] can be used to assist in analysing the results of mutation testing.

The purpose of coverage testing is to evaluate what parts of the SUT are exercised by the test suite. However, different meanings can be attributed to the concept of coverage, each requiring its own criterion. Traditionally, coverage testing is applied to software code, which can be structured as a graph [42]. In this context, several coverage criteria have been defined and classified according to their perspective. Some of the coverage criteria, such as node coverage (also called statement coverage in some literature [40]), edge coverage [43] or path coverage [44], measure the structural coverage of a graph. Other criteria, such as the definition-usage path coverage, focus on the flow of the data within the software [45]. A detailed description of the most relevant coverage criteria is presented in [46].

When coverage testing is performed on software code, this occurs through *instrumentation*, i.e., adding extra code (either statically, or dynamically at runtime [47]) which does not change the behaviour of the software, but collects some significant information [48] which is used to measure the coverage.

The idea to evaluate the coverage of an OWL ontology does not appear to have been explored in the past. Traditional coverage testing techniques must be revisited to allow such an analysis, primarily because there is no code which can be instrumented in an ontology. An ontology is essentially a knowledge base which can be used by software tools. Additionally, the peculiar structure of the ontology calls for new coverage criteria: although ontologies can certainly be represented as a graph, there is no standard way to do so, and nodes and edges can have different meanings in different representations. As stated in Sect. 3, OWL ontologies are made up of entities, and axioms which represent relations between entities. Both these components can be too generic and abstract to offer a clear coverage criterion.

The main focus of this paper is on mutation testing, and as such it does not intend to define a complete coverage testing approach for OWL ontologies. Rather, a basic coverage testing criterion for the limited scope of analysing the mutation testing results will be proposed. For the purposes of this paper, therefore, a coverage criterion which only takes into account the classes (which are generally the most relevant entities in an ontology) has been introduced. More specifically, the criterion will measure the coverage of the named classes, excluding the anonymous classes created by means of a restriction. This criterion will be called Named Class Coverage (NCC).

Preliminarily, the concept of visiting a named class can be expressed as follows.

Definition 1. *A test suite TS* ***visits*** *a given class C_i if at least one test $T \in TS$ is based on a query which retrieves C_i.*

This definition applies both when the SUT is the ontology itself and when it is a software which makes use of the ontology. In the former perspective, a test case will directly query the ontology and retrieve some entities and axioms. In the latter perspective, a test case may or may not exercise some code segment which queries the ontology.

Given this definition, the test requirement for class coverage is

$$TR = \{\text{visits class } C_1, \ldots, \text{visits class } C_n\}, \tag{1}$$

where n is the number of named classes in the ontology.

Definition 2 ***Named Class Coverage (NCC):*** *TR visits every named class asserted in the ontology.*

Therefore, the coverage of an ontology by a test suite TS is the percentage of named classes that are retrieved by the queries executed by the test suite.

Measuring this amount is not straightforward, and depends on the structure of the test suite. An example of how the coverage can be measured in a specific setup is shown in the following section.

The coverage of an ontology can be used to further derive test cases. In particular, the uncovered portions of the ontology are the ones that new test cases should explore. However, given the lack of literature in the topic of ontology coverage, there is currently no means to derive new test cases based on coverage.

4.1 An Application of NCC

To show a sample application of the NCC coverage criterion, the setup used in Sect. 5.4 will be used. The SUT will be the ontology itself, and the test suite will be made up of a set of SPARQL Protocol and RDF Query Language (SPARQL) queries.

A SPARQL query operates much like a query in a relational database: it accesses the knowledge base searching for content that matches the requested pattern, and produces an output in some format. However, the query needs to be modified to measure the coverage, because:

1. on one side, the SPARQL query may access more entities (including named classes) than those that are actually produced as output;
2. on the other side, during its search, the query will access some components of the ontology (e.g., other entities) that are not included in NCC.

Therefore, from the first perspective, the outputs of the query must be widened, to include all the classes which are searched but then left out of the report. From the second perspective, the query must be purged of all those elements of the ontology (e.g., labels and object properties) that are not class entities.

This normalization process is based on the following steps:

1. remove all FILTER operations (since they remove part of the results from the output);
2. only retrieve the named classes, ignoring any anonymous class;
3. remove search patterns based on label annotations and only retrieve the class IRIs;
4. change sub-queries into separate queries. For example, the MINUS operator executes a subquery which subtracts some results from the main query. However, these results are actually processed by the query, and if they contain class entities they must be accounted for, and not subtracted, to measure the coverage.

Additional changes were made for the ease of processing:

1. replace all blank nodes with identifiers;
2. purge the output format of the query of anything that is not a class entity;
3. split queries whose result format contains more than one result into a set of queries whose output format contains just one result. The queries thus generated will be identical, but each will output only one of the results of the original query;
4. remove namespace prefixes, using only full namespaces;
5. add a DISTINCT keyword (if not already present) to the query, to ensure that no duplicates are retrieved.

After these changes, each SPARQL query simply returns a set of named class entities. The union of all the result sets from the queries (ignoring any cross-query duplicates) is the complete set of named class entities involved by the test suite. Comparing this set to the total number of class entities gives a measure of the coverage.

5 Experiments

The proposed testing methodologies have been implemented and executed on several existing ontologies. This section describes the test platform, the reference ontologies and the results of the application of the mutation and coverage testing.

5.1 Experimental Setup

The implementation of the proposed mutation testing approach was done using Eclipse 4.5 (Mars) as a development environment. The programming language used is Java (Sun Java 1.8). The setup is platform-independent and has been successfully tested on Windows 7, Ubuntu Linux 14.04 and Mac OS X 10.10 machines, both at 32 and 64 bit.

The implementation is lightweight and only requires the following libraries, managed through Maven[1]:

- OWL API[2], for general processing of the ontologies;
- JFact[3], to parse inferred axioms within the ontologies;
- Apache Jena[4], to process the SPARQL query language.

The mutation testing tool, called Mutating OWLs, is available as a public Git repository[5]. The repository also contains the test ontologies described below.

5.2 Reference Ontologies

The proposed methodology has been executed on three different ontologies.

Data Protection. The data protection ontology has been introduced in [49, 50]. The European Union is currently undergoing a reform of the protection of personal data. The main legislative document of the reform is the General Data Protection Regulation (GDPR), which was very recently approved, introducing significant changes in the duties of the controller [51]. The ontology has been defined to describe the new reform; however, it does not aim at modeling the whole domain of data protection in the European Union, but only focuses on the requirements of the data controller.

The ontology is preliminary and subject to change, especially given that the reform is very recent and it lacks interpretation yes. It is mainly made up of hierarchical relations, and contains a number of object properties that relate the duties of the controller with the corresponding rights of the data subject.

Entities in the ontology are named using an auto-generated IRI, and labels contain the human-readable names.

Passenger Rights. The second ontology used as an experimental base has been introduced in [52,53] to describe the legal framework for flight incidents. In particular, the ontology addresses the perspective of the rights of the passenger.

This ontology has a more complex structure, and is split into three files. Since the import links were actually broken, some changes had to be made to the ontology to allow the OWL API to access local files. Specifically, the ontology had to be converted from Turtle syntax [54] to an XML serialization because of some limitations of OWL API in parsing non-XML syntaxes.

The naming convention differs from the previous ontology in that the IRIs are human-readable terms in English language, and no labels are used throughout the ontology.

[1] https://maven.apache.org/.
[2] http://owlapi.sourceforge.net/.
[3] http://jfact.sourceforge.net/.
[4] https://jena.apache.org/.
[5] https://github.com/guerret/lu.uni.owl.mutatingowls.

Pizza. Finally, the proposed methodology has been run against the well-known pizza ontology[6], which is the one provided as a standard example for OWL and Protégé tutorials. The naming convention used in this ontology is based on English-language identifiers for the entities, but entities also feature label annotations in Portuguese.

Summary. Table 2 displays a summary of the main features of the three ontologies used.

Table 2. Summary of the test ontologies.

	Data protection	Passenger rights	Pizza
Total number of axioms	848	541	940
Classes	88	89	100
Object properties	42	26	8
Data properties	3	31	0
Individuals	16	14	5
Subclass axioms	114	83	259

5.3 Experimental Results

The mutation operators defined in Sect. 3.2 have been applied to the three test ontologies, generating mutants for each. The total number of mutants per mutation operator is displayed in Table 3.

Some considerations are offered by the very structure of the three ontologies. For example, the data protection ontology, as mentioned earlier, uses auto-generated IRIs as identifiers, and labels for descriptive purposes. The pizza ontology uses English terms as identifiers, but entities also have Portuguese labels. Finally, the passenger rights ontology does not use label annotations. For this reason, the ERL and ECL operators do not generate any mutant in the latter. Similarly, no mutant is generated by the IAP, IAC and IRT operators in the passenger rights ontology because the individuals are not assigned to any class.

The data protection ontology makes a very limited use of data properties, so very few mutants are generated from the data property entity; the same is not true for the passenger rights entity, which has a significant number of data properties but less object properties. The pizza ontology does not have any data properties at all, and few object properties. However, the classes that make up the domain and range of some of the object properties have a large number of subclasses, hence many mutants from the ODC and ORC operators.

[6] http://protege.stanford.edu/ontologies/pizza/pizza.owl.

Table 3. Mutants by mutation operator.

Operator	Data protection	Passenger rights	Pizza
ERE	145	67	112
ERL	145	0	95
ECL	145	0	95
CAS	7102	886	8151
CRS	114	33	255
CSC	101	33	83
CAD	7084	886	7404
CRD	18	0	753
CAE	7076	886	8134
CRE	37	0	41
OND	41	10	6
ONR	37	8	7
ODR	41	10	6
ORD	37	8	7
ODP	31	8	6
ODC	228	54	250
ORP	31	5	7
ORC	126	22	253
ORI	0	0	0
DAP	1	29	0
DAC	3	3	0
DRT	2	13	0
IAP	12	0	0
IAC	30	0	0
IRT	12	0	10

5.4 Validation

The proposed approach was validated by testing the ontologies themselves and not an application running on top of them. Specifically, the validation was performed on the data protection ontology and on the pizza ontology. For the SUT to be an ontology, the simplest approach to test it is to have a set of SPARQL queries [55] which retrieve data from the ontology.

For the most part, the queries for the data protection ontology are the SPARQL representation of the competency questions that have been introduced in [50], to perform the assessment of that ontology.

On the other hand, unfortunately, no SPARQL test suite is readily available in literature for the pizza ontology. A set of queries exists as the test suite for an

alternative query language[7]. These could be used as a basis to assess the validity of the approach presented in this paper. The queries in that test suite were thus converted back to SPARQL. However, two more queries were added to the test suite, because the existing queries only search for very small parts of the ontology.

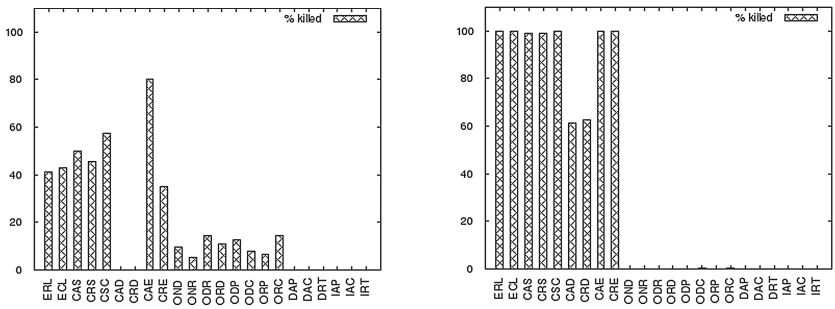
The complete experimental setup is available in the repository (see footnote 5).

To measure the coverage, the approach to measure the NCC of a set of SPARQL queries, as described in Sect. 4.1, was used. In both examples, this requires to slightly alter the structure of the SPARQL queries, as detailed in Sect. 4.1. Such a modification does not affect the content of the queries.

The NCC coverage of the set of SPARQL queries for the data protection ontology was measured as 62.50%. This means that the test suite (and, therefore, the competency questions from [50]) cover little more than half the named classes of the ontology. By measuring the coverage on the mutants, the results are highly variable: the minimum coverage is 31.03%, whereas the maximum one is 81.82%. The minimum coverage is reached on a single mutant of type ERE; the maximum coverage is reached on 17 mutants of type CAE and 9 mutants of type CAS. However, most of the mutants have the same coverage as the original SUT (62.50%).

On the other hand, the NCC coverage of the set of SPARQL queries for the pizza ontology was measured as 96.97%. This means that the test suite queries almost the whole set of named classes of the pizza ontology. With this SUT, The coverage on the mutants displays a very slight variation: the minimum coverage for the mutants is 95.96%, while the maximum is 97.96%. Specifically, all 95 mutants generated by the ERL operator (and only those) have the minimum coverage; whereas the maximum coverage is achieved by three of the mutants generated by the ERE operator.

The results of the validation is shown in Table 4, and a summary of killed mutants is shown in Fig. 1(a) and (b).



(a) Data protection ontology. (b) Pizza ontology.

Fig. 1. Overview of killed mutants.

[7] https://code.google.com/p/twouse/wiki/SPARQLASExamples.

Table 4. Results of the mutation testing.

Operator	Data protection			Pizza		
	Killed	Total	Percent	Killed	Total	Percent
ERE	61	145	42.07%	108	112	96.43%
ERL	60	145	41.38%	95	95	100%
ECL	62	145	42.76%	95	95	100%
CAS	3542	7102	49.87%	8073	8151	99.04%
CRS	52	114	45.61%	253	255	99.22%
CSC	58	101	57.43%	83	83	100%
CAD	0	7084	0%	4536	7404	61.26%
CRD	0	18	0%	471	753	62.55%
CAE	5678	7076	80.24%	8133	8134	99.99%
CRE	13	37	35.14%	41	41	100%
OND	4	41	9.76%	0	6	0%
ONR	2	37	5.41%	0	7	0%
ODR	6	41	14.63%	0	6	0%
ORD	4	37	10.81%	0	7	0%
ODP	4	31	12.90%	0	6	0%
ODC	18	228	7.89%	1	250	0.40%
ORP	2	31	6.45%	0	7	0%
ORC	18	126	14.29%	1	253	0.40%
DAP	0	1	0%	0	0	0
DAC	0	3	0%	0	0	0
DRT	0	2	0%	0	0	0
IAP	0	12	0%	0	0	0
IAC	0	30	0%	0	0	0
IRT	0	12	0%	0	10	0%
Total	9,584	22,599	42.41%	21,890	25,675	85.26%

A brief analysis of the results elicits some interesting considerations. First, it is clear that the test suites mainly address classes, with little attention to the properties, especially in the pizza ontology. Thus, in both cases, additional tests, especially for the object properties, are required. Also, concerning the classes, the tests in the data protection ontology mostly cover some specific branches of the hierarchy, while almost no tests search through other branches. Finally, some considerations can be done on the ontologies themselves. For example, by examining the live mutants generated by the ERE operator on the pizza ontology, it emerges that some object properties are not used by any of the SPARQL queries. Depending on the purposes of the ontology, this might suggest that those properties are irrelevant and would call for a structural change in the

ontology design. More significant insights could be offered by using richer test suites (possibly deriving the test cases from the coverage analysis), which are not currently available for the selected ontologies.

6 Conclusions and Future Work

The work presented in this paper extends and adapts some popular testing techniques from the software testing domain, namely mutation testing and coverage testing, to ontologies defined using the OWL language. The paper first gives a brief overview of the essentials of OWL ontologies. It then introduces a methodology and operators for mutation testing, and a possible approach to measure the coverage of an OWL ontology. Finally, it describes an implementation of the mutation and coverage testing techniques, and some basic experiments on previously-defined ontologies and SPARQL test suites.

The benefits of mutation testing are manifold: by analyzing the patterns of killed and alive mutants, testers can detect errors in the SUT and in the test suite. Equivalent mutants can help detect redundancies in the ontology, which may not be errors but still facilitate errors, for example when creating instances of the ontology.

On the other hand, combining mutation testing with coverage testing can assist in measuring the effectiveness of the test suite. In particular, measuring the coverage can help find the kinds of tests that need to be added to the test suite, and this in turn can lead to a higher percentage of killed mutants.

More in general, the extension of software engineering and testing approaches to ontologies and semantic knowledge bases can pave the way to the formalization of integrated design and testing patterns for semantics-based applications.

This work is at its initial stages, with many opportunities for future development. First off, the proposed methodology needs to be expanded to support a full test suite: a significant set of SPARQL queries, if the SUT is the ontology itself; or, if the SUT is an ontology-based software, testing it with its own test suite. The purpose would be to compare the outputs of the test suite when executed against the original ontology and against the mutants. In this phase, it is possible that the complexity of the mutation testing is excessive and causes performance problems, and it might be necessary to apply or develop algorithms designed to reduce the number of mutants.

Second, the mutation methodology can be improved, by extending it with additional mutation operators. With respect to the work presented in [56], additional mutation operators have been introduced, and these currently make up the bulk of the mutants generated. However, some features of the OWL language have not been exploited yet. For example, the mutation operators do not currently address annotations other than labels, or the value and cardinality constraints. Some of these OWL features can have a significant effect in the ontology definition, and mutants thus created might be useful in assessing the ontology.

Third, every mutation testing approach should be coupled with an algorithm to detect equivalent mutants, and the one proposed here makes no difference.

In particular, the newly-defined mutation operators (CAS, CAD and CAE) generate a very large number of mutants, possibly introducing performance issues. Identifying and removing equivalent mutants would then be of primary importance. In the specific domain of OWL ontologies, it is possible that the use of reasoners can provide an efficient means of detecting mutants.

Fourth, the mutation testing should take into account the peculiarities of ontology engineering. In particular, while the domain certainly imposes some constraints on the ontology developer, many decisions are based on discretionary choices, balancing different aspects such as human readability and efficiency of the ontology. Traditional mutation testing techniques might be extended to embrace these features, for example by separating those mutant operators that are likely to introduce errors in the domain (for example swapping a class with its parent) from those that simply change the ontology structure without making it inconsistent with the domain. If such a partition were possible, then mutation testing techniques could be used not only to detect errors in the design, but also to suggest different ontology architectures that the designer might overlook.

Finally, stretching along the line of the previous point, an extended mutation technique could be designed which alters the structure of the ontology. For example, there might be circumstances where using a hierarchical relationship (subclass axiom) might be an alternative to using an object property. An extended mutation technique that generates mutants based on a different structure of the ontology might offer a fast way to compare a wide number of ontology designs.

An even more significant amount of work would concern coverage testing. A very basic approach has been introduced in this work, which only takes into account OWL named classes, but measuring the coverage should involve much more than classes, e.g., addressing properties and individuals. Therefore, additional coverage criteria need to be defined.

Implementation-wise, OWL coverage testing also needs a lot of improvements. Specifically, due to the lack of instrumenting methodologies and tools for OWL ontologies, the coverage analysis currently requires to restructure the SPARQL queries so that it is possible to count the classes used. A more correct implementation would introduce instrumentation code that generates the coverage results. However, since SPARQL queries are not executable code but require a SPARQL engine to be run, the instrumentation should be performed on the latter. Since several SPARQL engines (including the one used in this work) have an open source implementation, such instrumentation is possible.

References

1. Quillian, M.R.: Word concepts: a theory and simulation of some basic semantic capabilities. Behav. Sci. **12**, 410–430 (1967)
2. World Wide Web Consortium (W3C): RDF 1.1 concepts and abstract syntax (2014)
3. World Wide Web Consortium (W3C): OWL 2 Web Ontology Language document overview, 2nd edn. (2012)

4. Antoniou, G., van Harmelen, F.: Web Ontology Language: OWL. In: Staab, S., Studer, R. (eds.) Handbook on Ontologies. International Handbooks on Information Systems, pp. 67–92. Springer, Heidelberg (2004)
5. Horrocks, I.: DAML+OIL: a description logic for the semantic web. Bull. Tech. Committee Data Eng. **25**, 4–9 (2002)
6. Horrocks, I.: What are ontologies good for? In: Küppers, B.O., Hahn, U., Artmann, S. (eds.) Evolution of Semantic Systems, pp. 175–188. Springer, Heidelberg (2013)
7. Rospocher, M., Serafini, L.: An ontological framework for decision support. In: Takeda, H., Qu, Y., Mizoguchi, R., Kitamura, Y. (eds.) JIST 2012. LNCS, vol. 7774, pp. 239–254. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37996-3_16
8. Kershenbaum, A., Fokoue, A., Patel, C., Welty, C., Schonberg, E., Cimino, J., Ma, L., Srinivas, K., Schloss, R., Murdock, J.W.: A view of OWL from the field: use cases and experiences. In: Cuenca Grau, B., Hitzler, P., Shankey, C., Wallace, E. (eds.) Proceedings of the Second Workshop on OWL: Experiences and Directions (OWLED), vol. 216. CEUR Workshop Proceedings (2006)
9. Blomqvist, E., Seil Sepour, A., Presutti, V.: Ontology testing - methodology and tool. In: Teije, A., Völker, J., Handschuh, S., Stuckenschmidt, H., d'Acquin, M., Nikolov, A., Aussenac-Gilles, N., Hernandez, N. (eds.) EKAW 2012. LNCS, vol. 7603, pp. 216–226. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33876-2_20
10. World Wide Web Consortium (W3C): OWL Web Ontology Language test cases (2004)
11. Wang, H., Horridge, M., Rector, A., Drummond, N., Seidenberg, J.: Debugging OWL-DL ontologies: a heuristic approach. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 745–757. Springer, Heidelberg (2005). doi:10.1007/11574620_53
12. García-Ramos, S., Otero, A., Fernández-López, M.: OntologyTest: a tool to evaluate ontologies through tests defined by the user. In: Omatu, S., Rocha, M.P., Bravo, J., Fernández, F., Corchado, E., Bustillo, A., Corchado, J.M. (eds.) IWANN 2009. LNCS, vol. 5518, pp. 91–98. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02481-8_13
13. McGuinness, D.L., Fikes, R., Rice, J., Wilder, S.: An environment for merging and testing large ontologies. In: Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), pp. 483–493 (2000)
14. Gangemi, A., Catenacci, C., Ciaramita, M., Lehmann, J.: Modelling ontology evaluation and validation. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 140–154. Springer, Heidelberg (2006). doi:10.1007/11762256_13
15. Burton-Jones, A., Storey, V.C., Sugumaran, V., Ahluwalia, P.: A semiotic metrics suite for assessing the quality of ontologies. Data Knowl. Eng. **55**, 84–102 (2005)
16. Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y., Liu, S.: Towards a complete OWL ontology benchmark. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 125–139. Springer, Heidelberg (2006). doi:10.1007/11762256_12
17. Guarino, N.: An overview of ontoclean. In: Staab, S., Studer, R. (eds.) Handbook on Ontologies. International Handbooks on Information Systems, 2nd edn, pp. 201–220. Springer, Heidelberg (2009)
18. Poveda-Villalón, M., Suárez-Figueroa, M.C., Gómez-Pérez, A.: Validating ontologies with OOPS!. In: Teije, A., Völker, J., Handschuh, S., Stuckenschmidt, H., d'Acquin, M., Nikolov, A., Aussenac-Gilles, N., Hernandez, N. (eds.) EKAW 2012. LNCS, vol. 7603, pp. 267–281. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33876-2_24

19. Poveda, M., Suárez-Figueroa, M.C., Gómez-Pérez, A.: Common pitfalls in ontology development. In: Meseguer, P., Mandow, L., Gasca, R.M. (eds.) CAEPIA 2009. LNCS, vol. 5988, pp. 91–100. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14264-2_10
20. Vrandečić, D., Gangemi, A.: Unit tests for ontologies. In: Meersman, R., Tari, Z., Herrero, P. (eds.) OTM 2006. LNCS, vol. 4278, pp. 1012–1020. Springer, Heidelberg (2006). doi:10.1007/11915072_2
21. Granitzer, M., Scharl, A., Weichselbraun, A., Neidhart, T., Juffinger, A., Wohlgenannt, G.: Automated ontology learning and validation using hypothesis testing. In: Wegrzyn-Wolska, K.M., Szczepaniak, P.S. (eds.) Advances in Intelligent Web Mastering. Advances in Soft Computing, vol. 43, pp. 130–135. Springer, Heidelberg (2007)
22. Lee, S., Bai, X., Chen, Y.: Automatic mutation testing and simulation on OWL-S specified web services. In: Proceedings of the 41st Annual Simulation Symposium (ANSS), pp. 149–156. IEEE (2008)
23. World Wide Web Consortium (W3C): OWL-S: Semantic markup for web services (2004)
24. Wang, Y., Bai, X., Li, J., Huang, R.: Ontology-based test case generation for testing web services. In: Proceedings of the 8th International Symposium on Autonomous Decentralized Systems (ISADS), pp. 43–50 (2007)
25. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer **11**, 34–41 (1978)
26. Hamlet, R.G.: Testing programs with the aid of a compiler. IEEE Trans. Softw. Eng. **SE-3**, 279–290 (1977)
27. Lipton, R.: Fault diagnosis of computer programs. Technical report. Carnegie Mellon University (1971)
28. Ammann, P., Offutt, A.J.: 5. In: Syntax-Based Testing, pp. 170–212. Cambridge University Press, Cambridge (2008)
29. Howden, W.E.: Weak mutation testing and completeness of test sets. IEEE Trans. Softw. Eng. **SE-8**, 371–379 (1982)
30. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Software Eng. **37**, 649–678 (2011)
31. Offutt, A.J.: A practical system for mutation testing: help for the common programmer. In: Proceedings of the International Test Conference (ITC). IEEE Computer Society, pp. 824–830 (1994)
32. Offutt, A.J., Untch, R.H.: Mutation 2000: Uniting the orthogonal. In: Wong, W.E. (ed.) Mutation Testing for the New Century. The Springer International Series on Advances in Database Systems, vol. 24, pp. 34–44. Springer, US (2001)
33. Bartolini, C., Bertolino, A., Marchetti, E., Parissis, I.: Data flow-based validation of web services compositions: perspectives and examples. In: Lemos, R., Giandomenico, F., Gacek, C., Muccini, H., Vieira, M. (eds.) WADS 2007. LNCS, vol. 5135, pp. 298–325. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85571-2_13
34. Ma, Y.S., Offutt, A.J., Kwong, Y.R.: Mujava: an automated class mutation system. Softw. Test. Verification Reliab. **15**, 97–133 (2005)
35. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Methodol. (TOSEM) **5**, 99–118 (1996)
36. Offutt, A.J., Hayes, J.H.: A semantic model of program faults. SIGSOFT Softw. Eng. Notes **21**, 195–200 (1996)

37. Mottu, J.-M., Baudry, B., Traon, Y.: Mutation analysis testing for model transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 376–390. Springer, Heidelberg (2006). doi:10.1007/11787044_28
38. Clark, J.A., Dan, H., Hierons, R.M.: Semantic mutation testing. In: Proceedings of the 3rd IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), pp. 100–109. IEEE (2010)
39. Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R., Wang, H.H.: The manchester OWL syntax. In: OWL: Experiences and Directions Workshop (OWLED) (2006)
40. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**, 366–427 (1997)
41. Yang, Q., Jenny Li, J., Weiss, D.M.: A survey of coverage-based testing tools. Comput. J. **52**, 589–597 (2009)
42. Ledgard, H.F., Marcotty, M.: A genealogy of control structures. Commun. ACM **18**, 629–639 (1975)
43. Huang, J.C.: An approach to program testing. ACM Comput. Surv. **7**, 113–128 (1975)
44. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Software Eng. **4**, 178–187 (1978)
45. Osterweil, L.J.: Data flow analysis as an aid in documentation, assertion, generation, validation, and error detection. Technical Report CU-CS-055-74, University of Colorado, Boulder, Colorado 80302 (1974)
46. Ammann, P., Offutt, A.J.: 2. In: Graph Coverage, pp. 27–103. Cambridge University Press, Cambridge (2008)
47. Tikir, M.M., Hollingsworth, J.K.: Efficient instrumentation for code coverage testing. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 86–96 (2002)
48. Ammann, P., Offutt, A.J.: 8. In: Building Testing Tools, pp. 268–279. Cambridge University Press, Cambridge (2008)
49. Bartolini, C., Muthuri, R.: Reconciling data protection rights and obligations: an ontology of the forthcoming EU regulation. In: Proceedings of the Workshop on Language and Semantic Technology for Legal Domain (LST4LD), Recent Advances in Natural Language Processing (RANLP) (2015)
50. Bartolini, C., Muthuri, R., Santos, C.: Using ontologies to model data protection requirements in workflows. In: Proceedings of the Ninth International Workshop on Juris-informatics (JURISIN), pp. 27–40 (2015). Extended version to be published in LNAI book
51. Reding, V.: The upcoming data protection reform for the European Union. International Data Privacy Law (2010)
52. Rodríguez-Doncel, V., Santos, C., Casanovas, P.: A model of air transport passenger incidents and rights. In: Proceedings of the 27th International Conference on Legal Knowledge and Information Systems (JURIX), pp. 55–60. IOS Press (2014)
53. Rodríguez-Doncel, V., Santos, C., Casanovas, P.: Ontology-driven legal support-system in the air transport passenger domain. In: Proceedings of the International Workshop on Semantic Web for the Law (SW4Law) (2014)
54. World Wide Web Consortium (W3C): RDF 1.1 Turtle (2014)
55. World Wide Web Consortium (W3C): Sparql query language for rdf (2008)
56. Bartolini, C.: Mutating OWLs: semantic mutation testing for ontologies. In: Proceedings of the workshop on domAin specific Model-based AppRoaches to vErificaTion and validaTiOn (AMARETTO), pp. 43–53 (2016)