Requirements Behaviour Analysis for Ontology Testing

Alba Fernández-Izquierdo(✉) and Raúl García-Castro

Ontology Engineering Group, Universidad Politécnica de Madrid, Madrid, Spain
{albafernandez,rgarcia}@fi.upm.es

Abstract. In the software engineering field, every software product is delivered with its pertinent associated tests which verify its correct behaviour. Besides, there are several approaches which, integrated in the software development process, deal with software testing, such as unit testing or behaviour-driven development. However, in the ontology engineering field there is a lack of clearly defined testing processes that can be integrated into the ontology development process. In this paper we propose a testing framework composed by a set of activities (i.e., test design, implementation and execution), with the goal of checking whether the requirements identified are satisfied by the formalization and analysis of their expected behaviour. This testing framework can be used in different types of ontology development life-cycles, or concerning other goals such as conformance testing between ontologies. In addition to this, we propose an RDF vocabulary to store, publish and reuse these test cases and their results, in order to allow traceability between the ontology, the test cases and their requirements. We validate our approach by integrating the testing framework into an ontology engineering process where an ontology network has been developed following agile principles.

Keywords: Ontology testing · Ontology requirements
Ontology development

1 Introduction

The increasing uptake of semantic technologies and ontologies has led during the past years to the study of new ontology development methodologies, from agile (e.g., [1,12]) to collaborative approaches (e.g., [15,17]). The majority of these methodologies take into account the importance of functional[1] ontology

This work is partially supported by the H2020 project VICINITY: Open virtual neighbourhood network to connect intelligent buildings and smart objects (H2020-688467) and by a Predoctoral grant from the I+D+i program of the Universidad Politécnica de Madrid.

[1] This term is borrowed from the Software Engineering field, in which functional requirements refer to the functionalities the software system should have.

C. Faron Zucker et al. (Eds.): EKAW 2018, LNAI 11313, pp. 114–130, 2018.
https://doi.org/10.1007/978-3-030-03667-6_8

requirements [16] which, written in natural language as competency questions [8], define the knowledge the ontology has to represent.

Nowadays, in software engineering it is inconceivable to deliver a software product without its pertinent tests which guarantee that it fulfils all its requirements. Besides, there are several approaches integrated into the software development process whose aim is to test the software. Unit testing [9], which validates that each unit of the software performs as designed, and behaviour-driven development [19], which focuses on the behaviour the software product is implementing, are examples of these approaches.

However, in ontology engineering there is a lack of clearly defined testing processes in order to be able to ascertain whether an ontology satisfies the requirements. Even though there are approaches to generate tests (e.g., [10,13]), they do not cover the entire testing workflow or are limited to checking for the presence of axioms, which is not enough to validate a requirement.

Inspired by the software engineering evaluation approaches, we propose a testing framework composed by a set of activities (i.e., test design, implementation and execution) with the aim of facilitating the generation and execution of tests associated to functional requirements of OWL ontologies. We propose to extract the behaviour of the requirements and to formalize it into test expressions. These test expressions are implemented into a set of axioms with the aim of validating if the ontology satisfies the intended knowledge produced by the requirements. The goal of this implementation is to solve the limitations of the actual testing approaches by analysing ontology behaviour in different situations to ascertain if the expected knowledge is present, absent or produces a conflict, rather than only checking whether an axiom is entailed by the ontology.

This proposed framework can be integrated into several ontology development life-cycles to support ontology development (e.g., to verify ontologies by users or ontology engineers) and also to carry out conformance analysis. In addition to this, we also propose an RDF vocabulary to represent the tests cases in order to provide traceability between them and the associated requirements.

The paper is organized as follows. Section 2 presents the related work on ontology testing. Section 3 presents the proposed testing activities and Sect. 4 describes the integration of these activities into ontology engineering workflows. Finally, Sect. 5 shows the evaluation of the approach and Sect. 6 presents the conclusions obtained and gives an overview on future work.

2 Related Work

Several approaches which defend the importance of verifying ontologies through their ontology requirements have been developed to the date. Each of these approaches focuses on some testing aspect: methodological background, test implementation, or traceability between the ontology and the tests.

Regarding the methodological background, Vrandevcic and Gangemi [18] introduced the idea of testing ontologies by borrowing ideas from software engineering, proposing techniques such as testing with axioms and negations

or formalizing competency questions. Another work presented by Peroni is SAMOD [12], an agile ontology development methodology that uses tests to validate the ontology. These two approaches are focused on methodological aspects but do not mention how to implement the tests or how to maintain traceability.

Concerning test implementation, Keet and Lawrynowicz proposed a test-driven development (TDD) of ontologies [10] in which the competency questions are formalized into axioms and added to the ontology if they are not present. Dealing also with test implementation, the OntologyTest tool [7] allows a user to define and execute a set of tests to check the functional requirements of an ontology; these tests are stored in an XML file for future reuse. Another approach to implement test cases is the one presented by Ren et al. [13]; in this work the authors use natural language processing to analyse competency questions written in controlled natural language from where they create competency question patterns that could be automatically tested in the ontology. Finally, Neuhaus introduced Scone[2], a tool for scenario-based ontology evaluation, which is based on Cucumber[3] and uses controlled natural language to define ontology scenarios which create mock individuals. Even though all these approaches are focused on test implementation, neither of them mention how to maintain traceability between the tests and the ontology nor do they describe the process to integrate ontology testing into ontology engineering methodologies in order to create the tests from the ontological requirements.

To conclude, Blomqvist et al. [2] presented an agile approach which includes a methodological background and introduces in rough outlines several types of tests concerned with the verification of the requirements implementation and the exposure to faults. However, this methodology neither explains how to implement these tests nor when each type of test should be used.

Even if all these works introduce testing through requirements, none of them proposes a complete testing framework which covers all the mentioned testing aspects. Moreover, the majority of these works do not allow the reuse of the tests, limiting the testing process only to a single ontology.

3 Ontology Testing Framework

This paper introduces an ontology testing framework to systematize the generation and execution of tests cases from functional requirements. In the literature, ontology testing approaches are usually divided into two activities, i.e., *test implementation* and *test execution*. In this approach we propose a new one, *test design*. The motivation of this new activity came up due to the ambiguity and assumptions inherent to the natural language [4] and to the fact that different people may be in charge of the design and implementation of tests. As a consequence, in this design activity the knowledge intended to be produced by every requirement is identified, e.g., from the requirement "*A device can have a status*" is expected a relation between two concepts in the ontology. From this

[2] https://bitbucket.org/malefort/scone.
[3] https://docs.cucumber.io/.

point on, we are going to call this expected knowledge as the *desired behaviour* of the requirement, which is concrete and unambiguous. In this design activity we provide a collection of test expressions according to the requirements behaviour, and in the test implementation activity we provide possible implementations for each one of the test expressions, which are ready to be executed on an ontology.

In this work we focus on the analysis of the behaviour of the ontology in different situations to verify that certain knowledge is modelled in the ontology, rather than simply checking the presence or absence of particular axioms using semantic reasoners. This is due to the fact that the use of semantic reasoners is not sufficient to validate if a requirement is satisfied. For instance, if we suppose that a requirement asks for a minimum cardinality of 1, the correspondent ontology already has the axiom $A \sqsubseteq B \sqcap \leq 2R.C$, and we check the presence of the axiom $A \sqsubseteq B \sqcap \leq 1R.C$ the reasoner will state that it is entailed, even though that is not what the requirement asks for. In this situation we would like to have a tool to determine that what is entailed in the ontology is not what the requirement claims. Because of this reason, it is needed to go beyond the result provided by the simple execution of reasoners. Another case where the checking for the presence of axioms using semantic reasoners is not sufficient is a situation in which an ontology has a large hierarchy of concepts. In this case, in which it is tedious to manually determine whether a certain concept belongs to that hierarchy, if some classes are not named the reasoner will not detect them in the hierarchy. The analysis of ontology behaviours aims to solve these problems.

In addition to the testing activities, we propose an RDF vocabulary[4] to store the generated test cases and to provide traceability between them and their associated requirements. To improve the readability, we assumed that A and B represent ontology classes, P represents an ontology property, *a*, *a1*, *b1* and *b2* represent individuals and *num* represents a numerical value.

3.1 Test Design

During this activity the desired behaviour of each requirement is extracted. In order to carry out this extraction, we provide a set of possible types of requirements according to their desired behaviour. Besides, each of these types of requirements is associated with a test expression, which represents the desired behaviour in a formal language based on the OWL Manchester Syntax[5]. In order to identify the different types of requirements, we analysed the 248 requirements of the following ontologies[6]: the VICINITY ontologies[7], the Video Game ontology [11] and the SAREF ontology[8]. We extracted the behaviour of each of these requirements and selected the ones that appear more than once or that we expect

[4] https://w3id.org/def/vtc.
[5] https://www.w3.org/TR/owl2-manchester-syntax/.
[6] At the time of writing, the authors only had access to the requirements of these ontologies.
[7] http://vicinity.iot.linkeddata.es/vicinity/.
[8] https://w3id.org/saref.

to appear frequently. Table 1 shows the identified types of requirements, their description and their corresponding test expressions. It is worth noting that, even though the greater part of the analysed requirements was categorized with only one type, a requirement could be categorized with more than one and, therefore, could be associated to more than one test expression.

Table 1. Types of requirements according to their behaviour

	ID	Type of requirement	Description	Test expression
Class-related	T1	Equivalence	Equivalence between two classes that have the same intention	A EquivalentTo B
	T2	Subsumption	Definition the relation between the class and the (super)class it belongs to. This subsumption is strict, the two classes cannot be equivalent	A SubClassOf B
	T3	Disjointness	Definition of two disjoint concepts	A disjointWith B
Property-related	T4	Property between two concepts	Definition of a property between two concepts	P Domain A, P Range B, A P B
	T5	Symmetry	A property must be symmetric, this means, the property has itself as an inverse	A Symmetric(P) B
	T6	Maximum cardinality	Definition of the maximum cardinality of a given property between two concepts	A SubClassOf P max [num] B
	T7	Minimum cardinality	Definition of the minimum cardinality of a given property between two concepts	A SubClassOf P min [num] B
	T8	Exact cardinality	Definition of the cardinality between two concepts	A SubClassOf P exactly [num] B
	T9	Intersection	Definition of an intersection between concepts with a cardinality	A SubClassOf P min/max/exactly [num] (B and C)
Individual-related	T10	Definition of an individual	Definition of an individual of a given type	a type A

The output of this activity is an RDF document where the test cases are stored using the proposed testing vocabulary. In this vocabulary, each test case design stores the associated requirement URI, the description of the requirement, and the desired behaviour specified by the test expressions.

Listing 1.1 shows an example of a test case design generated from the requirement which states "*An IoT gateway is a digital entity*". This requirement is categorized with one requirement type: Subsumption (T2). Because of the fact that this test does not have URIs related to the ontology in which the test cases are going to be executed, it can be reused in other ontologies. To improve the readability of the paper, Table 2 shows the prefixes and their associated namespaces that are used through the paper.

Listing 1.1. Example of test case design

```
:testDesignPlatform2 a vtc:TestCaseDesign;
        vtc:isRelatedToRequirement vicinity:platform2;
        dc:description "An IoT gateway is a digital entity";
        vtc:desiredBehaviour "<Gateway> subClassOf <DigitalEntity>".
```

3.2 Test Implementation

In order to implement the tests to verify if a desired behaviour is satisfied, we propose a procedure were each test expression is formalized into a precondition, a set of auxiliary term declarations and a set of assertions to check the behaviour. During this procedure it is also carried out a mapping between the term identified in the test design and the actual term in the ontology where the ontology is going to be executed. The testing framework proposed in this work provides implementation for every test expression identified in Table 1.

Table 2. Summary of the prefixes used through the paper

Prefix	Namespace
core	http://iot.linkeddata.es/def/core
dc	http://purl.org/dc/terms/
vicinity	http://vicinity.iot.linkeddata.es/vicinity/re-quirements/report-core.html
vtc	http://w3id.org/def/vtc

The **precondition** is a SPARQL query which checks whether the terms involved in the ontology requirement are defined in the ontology. In order to execute the tests, these terms need to be declared in the ontology. Otherwise, the test fails and the requirement is not satisfied.

The **axioms to declare auxiliary terms** are a set of temporary axioms added to the ontology to declare the auxiliary terms needed to carry out the assertions. After the addition of these axioms the reasoner is executed and, in order to be able to check the behaviour, the ontology needs to be consistent.

Finally, the **assertions to check the behaviour** are a set of pairs of axioms and expected results that represent different ontology scenarios. For each pair, the axiom is temporary added to the ontology to force a scenario, after which the reasoner is executed. The expected result determines if the ontology status (i.e., inconsistent ontology, unsatisfiable class or consistent ontology) after the addition is the expected one in case the requirement was satisfied. If all the status concur with the expected status, then the requirement is satisfied.

The output of this activity is an RDF document where the test cases are stored using the proposed vocabulary. In this vocabulary, each test case implementation stores the associated test design; the test preparation, which represents auxiliary terms declaration; and the corresponding test assertions. An excerpt of a test case is shown in Listing 1.2. Due to the lack of space the figure only shows the test precondition, which verifies that the classes involved in the test exist in the ontology; the test preparation, which adds the auxiliary terms needed for the execution of the test; and one of the assertions, which adds a class that will be wrong if the ontology satisfies the requirement.

Listing 1.2. Example of test case implementation

```
:testImplPlatform2 a vtc:TestCaseImplementation;
   vtc:isRelatedToDesign :testDesignPlatform2;
   vtc:precondition "ASK{ Class(core:Gateway), Class(core:DigitalEntity)}";
   vtc:hasPreparation :preparation1;
   vtc:hasAssertion [:assertion1; :assertion2; :assertion3].
:preparation1 a vtc:TestPreparation;
   dc:description "Declaration of the auxiliary terms";
   vtc:testAxioms """:NoGateway rdf:type owl:Class;
                              owl:complementOf core:Gateway.
                     :NoDigitalEntity rdf:type owl:Class;
                              owl:complementOf core:DigitalEntity. """.
:assertion1 a vtc:TestAssertion;
   dc:description "Test assertion 1";
   vtc:testAxioms """core:GatewayNoDigitalEntity rdf:type owl:Class;
                             rdfs:subClassOf  core:Gateway;
                             rdfs:subClassOf  :NoDigitalEntity.""";
   vtc:hasAssertionResult vtc:Unsatisfiable .
```

3.2.1 Class-Related Test Expressions

Table 3 shows the implementations to verify equivalence (T1), subsumption (T2), and disjointness (T3) between two concepts.

To check **equivalence** between two concepts, we define a set of auxiliary terms, i.e., the classes that complement A ($\neg A$) and B ($\neg B$). After their definition, we define a set of assertions that force the ontology to present unsatisfiable classes or inconsistencies. The first one, associated to axiom 'E 3' in Table 3, generates a class A' that is defined as a subclass of class B and $\neg A$. If the ontology satisfies the requirement, this addition causes an unsatisfiable class due to the fact that the reasoner would infer that A' is subclass of A and $\neg A$. The second assertion, associated to axiom 'E 4', generates a class A' that is defined as a subclass of class A and $\neg B$. If the ontology satisfies the requirement, this addition causes an unsatisfiable class due to the fact that the reasoner would infer that A' is subclass of B and $\neg B$. The last assertion, associated to axiom 'E 5', generates a class A' that is defined as a subclass of class A and B. If the

Table 3. Test implementation for class-related test expressions

	Preconditions	Axioms to declare auxiliary terms	Assertions to test the ontology behaviour	
			Axiom	Expected status after adding the axiom
T1	Class A and class B exist	(E 1) Declaration of ¬A (E 2) Declaration of ¬B	(E 3) A' ⊑ ¬A ⊓ B	Unsatisfiable class
			(E 4) A' ⊑ A ⊓ ¬B	Unsatisfiable class
			(E 5) A' ⊑ A ⊓ B	Consistent ontology
T2	Class A and class B exist	(S 1) Declaration of ¬A (S 2) Declaration of ¬B	(S 3) A' ⊑ ¬A ⊓ B	Consistent ontology
			(S 4) A' ⊑ A ⊓ ¬B	Unsatisfiable class
			(S 5) A' ⊑ A ⊓ B	Consistent ontology
T3	Class A and class B exist	(D 1) Declaration of ¬A (D 2) Declaration of ¬B	(D 3) A' ⊑ ¬A ⊓ B	Consistent ontology
			(D 4) A' ⊑ A ⊓ ¬B	Consistent ontology
			(D 5) A' ⊑ A ⊓ B	Unsatisfiable class

ontology satisfies the requirement, this assertion causes a consistent ontology due to the fact that there is no problem if *A'* is subclass of *A* and *B*.

We follow the same procedure for each of the test expressions, defining scenarios which cause different behaviours in the ontology.

In the case that the requirement involves **subsumption** between concepts and the ontology meets the requirement, axiom 'S 4' in Table 3 causes an unsatisfiable class. Axioms 'S 3' and 'S 5' are expected to entail consistent ontologies.

Finally, if the requirement involves **disjoint** classes and the ontology satisfies the requirement, axiom 'D 5' in Table 3 causes an unsatisfiable class. Axioms 'D 3' and 'D 4' are expected to entail consistent ontologies.

3.2.2 Property-Related Test Expressions

By the same token, Table 4 shows the implementations related to properties between concepts (T4), symmetry (T5), cardinalities (T6, T7, T8), and intersection (T9).

In order to check a **property between two concepts** *A* and *B*, a new individual is added to the ontology. The assertion associated to axiom 'Pst 6' in Table 4 defines a link between two individuals. If the range is defined, this assertion causes an inconsistent ontology due to the fact that the reasoner infers that one individual is of type *B* and its complement.

In order to check **symmetry**, the assertions add two properties between different individuals. The assertion associated to axiom 'Sy 6' defines a property between individuals that does not cause any inconsistency. However, the assertion associated to axiom 'Sy 7' defines a property between individuals that should not satisfy the constraint and causes an inconsistent ontology.

In order to check cardinality, the assertions define axioms that add new cardinality constraints to the ontology. Depending on the type of cardinality, different axioms cause an unsatisfiable class. If the requirement involves a **maximum cardinality** and the ontology satisfies the requirement, axiom 'Max 3' causes an unsatisfiable class. However, if the requirement involves a **minimum cardinality** and the ontology satisfies the requirement, axiom 'Min 2' causes an unsatisfiable class. Finally, if the requirement involves an **exact cardinality** and the ontology satisfies the requirement, axioms 'Ex 2' and 'Ex 3' cause an unsatisfiable class.

Finally, in order to check **intersection** between B and C, the first assertion follows the same principles than the maximum cardinality tests. In addition, assertions 'I 10' and 'I 11' force an axiom which does not satisfy the cardinality nor does it consider the intersection. These last assertions should lead to an consistent ontology, due to the fact that although they do not satisfy the cardinality constraint they do not satisfy the intersection.

3.2.3 Individual-Related Test Expressions

Regarding the individual-related test expressions, Table 5 defines the implementation for the test related to the **definition of an individual** *a* of type *A* (T10).

Table 4. Test implementation for the property-related test expressions

	Preconditions	Axioms to declare auxiliary terms	Assertions to test the ontology behaviour	
			Axioms	Expected status after adding the axiom
T4	Class A, class B and property P exist	(Pst 1) Declaration of ¬B (Pst 2) Assertion A' ⊑ A (Pst 3) Assertion A'(a1) (Pst 4) Assertion ¬B(nob1) (Pst 5) Assertion A' ⊑ ∃P.{nob1}	(Pst 6) Assertion P(a1, nob1)	Inconsistent ontology
T5	Class A, class B and property P exist	(Sy 1) Assertion B(b1) (Sy 2) Declaration A' ⊑ A (Sy 3) Assertion A' ⊑ ∀P.{b1} (Sy 4) Assertion A'(a1) (Sy 5) Assertion B(b2)	(Sy 6) Assertion P(a1, b1)	Consistent ontology
			(Sy 7) Assertion P(b2, a1)	Inconsistent ontology
T6	Class A, class B and property P exist	(Max 1) Declaration of A' ⊑ A	(Max 2) Assertion A' ≤ (num-1)R.B	Consistent ontology
			(Max 3) Assertion A' ≥ (num+1)R.B	Unsatisfiable class
			(Max 4) Assertion A' ≤ (num)R.B	Consistent ontology
			(Max 5) Assertion A' ≥ (num)R.B	Consistent ontology
T7	Class A, class B and property P exist	(Min 1) Declaration of A' ⊑ A	(Min 2) Assertion A' ≤ (num-1)R.B	Unsatisfiable class
			(Min 3) Assertion A' ≥ (num+1)R.B	Consistent ontology
			(Min 4) Assertion A' ≤ (num)R.B	Consistent ontology
			(Min 5) Assertion A' ≥ (num)R.B	Consistent ontology
T8	Class A, class B and property P exist	(Ex 1) Declaration of A' ⊑ A	(Ex 2) Assertion A' ≤ (num-1)R.B	Unsatisfiable class
			(Ex 3) Assertion A' ≥ (num+1)R.B	Unsatisfiable class
			(Ex 4) Assertion A' ≤ (num)R.B	Consistent ontology
			(Ex 5) Assertion A' ≥ (num)R.B	Consistent ontology
T9	Class A, class B and property P exist	(I 1) Declaration of A' ⊑ A (I 2) Declaration of ¬B (I 3) Declaration of ¬C	(I 4) Assertion A' ≤ (num-1).B ⊓ C	Consistent ontology
			(I 5) Assertion A' ≥ (num+1)R.B ⊓ C	Unsatisfiable class
			(I 6) Assertion A' ≤ (num)R.B ⊓ C	Consistent ontology
			(I 7) Assertion A' ≥ (num)R.B ⊓ C	Consistent ontology
			(I 8) Assertion A' ≤ (num)R.B ⊓ C	Consistent ontology
			(I 9) Assertion A' ≥ (num)R.B ⊓ C	Consistent ontology
			(I 10) Assertion A' ≥ (num+1)R.B	Consistent ontology
			(I 11) Assertion A' ≥ (num+1)R. C	Consistent ontology

To check it, axiom 'Id 3' first identifies if there is a problem with the definition of the individual. To conclude, axiom 'Id 4' declares that the individual a is of type complement of A; in this case the assertion causes an inconsistency, due to the fact that an individual cannot be of type A and its complement.

Table 5. Test implementation for the individual-related test expression

	Preconditions	Axioms to declare auxiliary terms	Assertions to test the ontology behaviour	
			Axioms	Expected status after adding the axiom
T10	Class A and individual a1 exist	(Id1) Declaration of ¬A (Id2) Declaration of B	(Id3) Assertion B(a1)	Consistent ontology
			(Id4) Assertion ¬A(a1)	Inconsistent ontology

3.3 Test Execution

The test execution activity consists of three parts: the execution of the query which represents the preconditions, the addition of the axioms which declare the auxiliary terms, and the addition of the assertions. After the addition of each axiom, the reasoner is executed to report the status of the ontology. The addition of the auxiliary axioms needs to always lead to a consistent ontology. However, in the case of the assertions, the agreement between the reasoner status after the addition of all the axioms and the status indicated in the test implementation determines whether the ontology satisfies the desired behaviour.

We distinguish three possible results, i.e., *undefined*, if the ontology does not pass the preconditions; *passed*, if the ontology passes the preconditions and the results of the assertions are the expected ones; and *not passed*, if the ontology passes the preconditions but the results of the assertions are not the expected ones. The separation between *not passed* and *undefined* tests distinguishes between an incorrect behaviour of the ontology, where the constraints or characteristics of the tested concepts are not defined, and an absent behaviour, where the tested concepts are not defined. Algorithm 1 summarizes the steps needed to execute each test case. If the test case is passed, then the requirement is satisfied; otherwise, the requirement is not satisfied. Moreover, the *not passed* result implies that the requirement is not correctly implemented, while and *undefined* result implies that the requirement is not taken into account in the ontology implementation.

Algorithm 1. Test case execution

```
Data: Ontology and test case implementation
Result: Test case result
if precondition = true then
    add(ontology, auxiliary terms);
    if checkOntologyStatus() = consistent then
        for assertion in assertions do
            add(ontology, assertion.axioms);
            if checkOntologyStatus() ≠ assertion.result
            then
                result = not passed;
                exit loop;
            end
            remove(ontology, assertion.axioms);
            result = passed;
        end
    else
        result = not passed;
    end
    remove(ontology, auxiliary terms);
else
    result = undefined;
end
```

The output of this activity is an RDF document where the results of each test case are stored using the proposed vocabulary. In this vocabulary, each test case result stores the URI of the ontology that is tested, the test implementation and the result of the execution on the ontology.

Listing 1.3. Example of test case result

```
:testResultPlatform2 a vtc:TestCaseResult;
     vtc:hasExecution :execution1;
     vtc:testResult vtc:Undefined.
:execution1 a vtc:Execution;
     vtc:executedOn <http://iot.linkeddata.es/def/core/ontology.ttl>;
     vtc:isRelatedToImplementation :testImplPlatform2 .
```

4 Testing process

The proposed testing activities can be used in several *test-last* ontology development life-cycles, such as in waterfall [5] or in agile [12] ones. In the case of waterfall ontology development, the tests are generated and executed at the end of the development process to validate it. On the other hand, in an agile approach the development of the ontology is incremental based on development iterations or sprints and the tests are generated and executed after each iteration.

Moreover, the testing activities can also be integrated into *test-first* approaches, such as Test-Driven Development (TDD), where the tests are

generated before the ontology implementation in order to guide the development. Inspired by software engineering, we support another *test-first* approach: Behaviour-Driven Development (BDD). This approach, introduced in the Scone project[9], focuses on the behaviour the ontology needs to implement. In software engineering, BDD is focused on defining specifications of the behaviour of the system, in a way that they can be automated [14]. The main goal of BDD, which is generally regarded as the evolution of TDD, is to get executable specifications of a system that can be used by the users. Figures 1 and 2 depict the workflows of these *test-last* and *test-first* approaches. The application of this approach to ontology engineering may help ontology engineers to be more conscious about the ontology behaviour expected by the users.

In addition to the integration into the ontology development, the proposed testing activities can also be used for other goals, such as to verify the conformance that two ontologies have according to their requirements or to execute regression tests. Because the test design is separated from the test implementation, the test cases design can be reused over different ontologies instead of being generated from scratch.

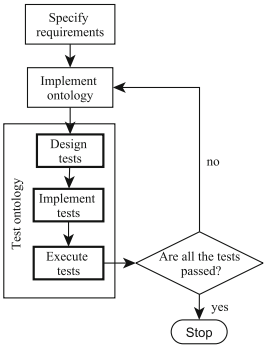

Fig. 1. Test-last approach

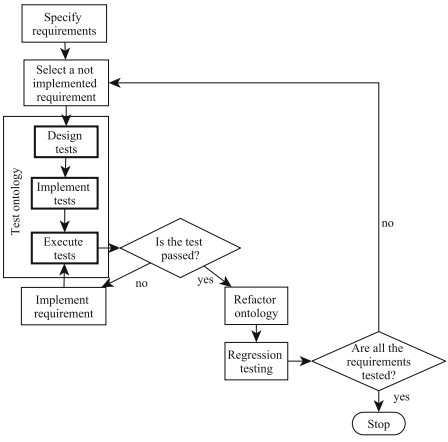

Fig. 2. Test-first approach

5 Evaluation

To provide an assessment of the validity of the proposed testing framework and its usability in an ontology development project, an empirical analysis has been carried out using three different ontologies, being the VICINITY Core[10]

[9] https://bitbucket.org/malefort/scone.
[10] http://iot.linkeddata.es/def/core/.

(Core[11]), the Web of Things[12] (WoT) and the WoT mappings[13] (Mappings) ontologies, which are currently under development in the VICINITY project[14]. To perform such assessment, we have integrated the proposed testing activities into the ontology development process, which was iterative and followed agile principles. Altogether, we gathered 123 ontology requirements, from which 16 were associated to the WoT ontology, 92 to the Core ontology and 15 to the Mappings ontology.

We generated test cases for those ontology requirements that were planned for the different sprints and analysed them to obtain information about their categorization. Table 6 shows the percentage of requirements categorized with one or more of the requirement types identified in Sect. 3. This table gives us information about the complexity of the requirements. We found that most of the requirements are related to only one type of requirement, i.e., relation between concepts.

Table 6. Percentage of requirements whose desired behaviour is categorized with one or more types

Requirement categorization	Core	WoT	Mappings	Total
One type	87%	94%	100%	89%
Two types	13%	6%	0%	11%
More than two types	0%	0%	0%	0%

Table 7 shows the number of requirements which belong to each requirement type. We found that the requirements related to hierarchies and to relations between concepts are the most common requirements in our analysed ontologies.

The execution of the test cases following this approach allows us to be aware of the **test results**, including the number of tests that are passed, not passed and undefined. In addition to this, the storage of the test cases in RDF with metadata permits an automated execution of test cases as well as maintaining the traceability between the test cases, the ontology and the requirements. Because of this traceability, we are able to calculate metrics such as the percentage of tested terms and the percentage of formalized requirements, which can provide us with an outlook about the situation of the testing process. This information is useful for the developers to be aware about which requirements are fulfilled by the ontologies and which ones are not implemented yet, as well as about which ontology terms are present in the tests. To calculate the tested terms we defined a metric called **tested terms coverage** (TTCOV), which is calculated using the expression

$$TTCOV(S, O) = \frac{NTestedT(S)}{NT(O)} \quad (1)$$

[11] During the development of this work part of the VICINITY Core ontology was transferred to a new ontology.
[12] http://iot.linkeddata.es/def/wot/.
[13] http://iot.linkeddata.es/def/wot-mappings/.
[14] http://vicinity2020.eu/vicinity/.

Table 7. Number of requirements of each type in the analysed ontologies

Type of requirement	Core	WoT	Mappings	Total
T1 - Equivalence	0	0	0	0
T2 - Subsumption	35	5	1	41
T3 - Disjointness	0	0	0	0
T4 - Relation between two concepts	86	10	9	105
T5 - Symmetry	4	0	0	4
T6 - Maximum cardinality	0	0	1	1
T7 - Minimum cardinality	2	1	0	3
T8 - Exact cardinality	0	0	1	1
T9 - Intersection	1	0	1	1
T10 - Definition of an individual	11	0	0	11

where *NTested(S)* refers to the number of different terms in the set of tests S and *NT(O)* refers to the number of terms defined in the ontology O.

To calculate the tested requirements we defined a metric called **formalized requirements coverage** (FRCOV), which is calculated using the expression

$$FRCOV(R,S) = \frac{NR(R)}{NTests(S)} \tag{2}$$

where $NR(R)$ refers to the number of identified requirements and $NTests(S)$ refers to the number of tests cases generated.

Table 8 summarizes the results obtained after the execution of the test cases in the last sprint of each ontology. All the requirements, their test cases and results are published in the VICINITY portal[15]. The results show that, even though the majority of the requirements are passed, there are several *undefined* tests in the Core ontology. This is due to the fact that there are several terms identified in the requirements which are not yet declared in the ontologies because they have not been planned yet for any sprint. Additionally, the results also

Table 8. Metrics extracted from the test cases in their last sprint

Ontology	Test results			Tested terms	Formalized requirements
	Passed	Not passed	Undefined		
Core	59%	17%	24%	41%	100%
WoT	94%	6%	0%	53%	100%
Mappings	100%	0%	0%	83%	100%
Total	68%	14%	18%	49%	100%

[15] http://vicinity.iot.linkeddata.es/vicinity/testing.html.

show that both the Core and the WoT ontologies have requirements that are not passed. Table 8 also determines that the tested terms do not exceed the 83%; this result is normal because there are terms that are not defined in the requirements. These terms can be created from the addition of ontology design patterns [6] or from the reuse of terms from other ontologies.

6 Conclusions and Future Work

In this paper we provide a testing framework composed of a set of activities that can be integrated into different ontology development life-cycles. This framework also provides a collection of test expressions to determine the desired behaviour of the requirements. These test expressions were defined after an analysis of 248 requirements from different ontologies. If more requirements with new behaviours are available, the set of test expressions will be extended to support them.

In addition to this, the storage of the tests in an RDF document allows us to extract different metrics, such as the already mentioned TTCOV and FRCOV, with the aim of better monitoring the ontology testing process. We expect that adopting testing activities in the development process will allow ontology engineers and users to be aware about the completeness of ontologies regarding their requirements. Moreover, these testing activities can also be helpful for analysing ontology conformance.

Future work will be directed to a more rigorous analysis of the requirement types. We plan to conduct a lexico-syntactic analysis of the requirements, based on the work presented by Daga et al. [3], in order to be able to identify more enriched test expressions. Furthermore, due to the fact that the test cases analyse the ontology status in different scenarios by adding several axioms, future work will also be directed to support the identification of the reason of why a test is failing. This would make the proposed testing framework helpful not only to verify if all the requirements are satisfied, but also to explain what is left for the ontology to fulfil the requirement.

Finally, we plan to analyse the feasibility and the benefits of the BDD approach applied to ontologies; we consider that this approach may help ontology engineers to provide ontologies more aligned with user expectations. Additionally, in this work we focused on OWL ontologies, and we intend to provide support for ontologies in other languages, e.g., RDF Schema.

References

1. Auer, S.: The RapidOWL Methodology - towards agile knowledge engineering. In: Proceedings of the IEEE International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2006), pp. 352–357 (2006)
2. Blomqvist, E., Seil Sepour, A., Presutti, V.: Ontology testing - methodology and tool. In: ten Teije, A., et al. (eds.) EKAW 2012. LNCS, vol. 7603, pp. 216–226. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33876-2_20

3. Daga, E., et al.: NeOn D2.5.2 Pattern based ontology design: methodology and software support
4. Dennis, M., van Deemter, K., Dell'Aglio, D., Pan, J.Z.: Computing authoring tests from competency questions: experimental validation. In: d'Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10587, pp. 243–259. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68288-4_15
5. Fernández-López, M., Gómez-Pérez, A., Juristo, N.: Methontology: from ontological art towards ontological engineering. In: Proceedings of the Ontological Engineering AAAI 1997 Spring Symposium Series, pp. 33–40 (1997)
6. Gangemi, A., Presutti, V.: Ontology design patterns. In: Staab, S., Studer, R. (eds.) Handbook on Ontologies, pp. 221–243. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-92673-3
7. García-Ramos, S., Otero, A., Fernández-López, M.: OntologyTest: a tool to evaluate ontologies through tests defined by the user. In: Omatu, S., et al. (eds.) IWANN 2009. LNCS, vol. 5518, pp. 91–98. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02481-8_13
8. Grüninger, M., Fox, M.S.: Methodology for the Design and Evaluation of Ontologies (1995)
9. Hamill, P.: Unit Test Frameworks: Tools for High-quality Software Development. O'Reilly Media Inc., Sebastopol (2004)
10. Keet, C.M., Ławrynowicz, A.: Test-driven development of ontologies. In: Sack, H., et al. (eds.) ESWC 2016. LNCS, vol. 9678, pp. 642–657. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34129-3_39
11. Parkkila, J., et al.: An ontology for videogame interoperability. Multimedia Tools Appl. **76**(4), 4981–5000 (2017)
12. Peroni, S.: A simplified agile methodology for ontology development. In: Dragoni, M., Poveda-Villalón, M., Jimenez-Ruiz, E. (eds.) OWLED/ORE 2016. LNCS, vol. 10161, pp. 55–69. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-54627-8_5
13. Ren, Y., Parvizi, A., Mellish, C., Pan, J.Z., van Deemter, K., Stevens, R.: Towards competency question-driven ontology authoring. In: Presutti, V. (ed.) ESWC 2014. LNCS, vol. 8465, pp. 752–767. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07443-6_50
14. Solis, C., Wang, X.: A study of the characteristics of behaviour driven development. In: Proceedings on the EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2011), pp. 383–387 (2011)
15. Suárez-Figueroa, M.C., Gómez-Pérez, A., Fernández-López, M.: The NeOn methodology for ontology engineering. In: Suárez-Figueroa, M.C., Gómez-Pérez, A., Motta, E., Gangemi, A. (eds.) Ontology Engineering in a Networked World, pp. 9–34. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-24794-1_2
16. Suárez-Figueroa, M.C., Gómez-Pérez, A., Villazón-Terrazas, B.: How to write and use the ontology requirements specification document. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2009. LNCS, vol. 5871, pp. 966–982. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05151-7_16
17. Sure, Y., Erdmann, M., Angele, J., Staab, S., Studer, R., Wenke, D.: OntoEdit: collaborative ontology development for the semantic web. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 221–235. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-48005-6_18

18. 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). https://doi.org/10.1007/11915072_2
19. Wynne, M., Hellesoy, A., Tooke, S.: The Cucumber Book: Behaviour-Driven Development for Testers and Developers. Pragmatic Bookshelf, Raleigh (2017)