OntologyTest: A Tool to Evaluate Ontologies through Tests Defined by the User

Sara García-Ramos, Abraham Otero, and Mariano Fernández-López

Universidad San Pablo CEU, Escuela Politécnica Superior, Urbanización Montepríncipe s/n, 28668 Boadilla del Monte, Madrid, Spain
{s.garcia,aotero,mfernandez.eps}@ceu.es

Abstract. The ontology evaluation utilities that are currently available allow the user to check the internal consistency of an ontology, its syntactical correctness and, at most, the fulfillment of some philosophical constraints related to rigidity or identity. However, there is no contribution in the ontology evaluation field that proposes a method to dynamically test ontologies with regard to their functional specification. Thus, no software for this task has been built until now. This paper presents a tool, OntologyTest, designed to overcome this drawback. The tool allows the user to define a set of tests to check the ontology's functional requirements, to execute them, and to inspect the results of the execution. The whole set of tests (or a particular test) can be executed at any time; thus it simplifies the testing of ontology both during its development and during its evolution.

Keywords: ontology, test, OWL DL, SPARQL, ontology evaluation.

1 Introduction

Although the term *ontology* is defined and explained in different ways, the definition that seems to be the most accepted is from Gruber [1]: *"an ontology is an explicit specification of a conceptualization"*, where *conceptualization* refers to *"the objects, concepts, and other entities that are assumed to exist in some area of interest and the relationships that hold among them (…)"*. If we assume that the conceptualization is made explicit through an implementation language –e.g. OWL [2]– then the definition can be reformulated as *an ontology is the axiomatization, in a formal language endowed with formal semantics, of a theory that makes a conceptualization explicit* [1].

As any other resource used in software applications, ontologies should be evaluated during their development and before their (re)use in other ontologies or applications [3]. However, no contribution in ontology evaluation proposes a method to dynamically test ontologies with regard to their functional specification [4, 5], neither has software for this task been built until now. The purpose of OntologyTest is to overcome this drawback. This utility allows the user to define tests to check the ontology functional requirements. Each test comprises an optional set of instances, a

[1] Inspired from Jesús Bermúdez de Andrés and Mariano Fernández López's definition elaborated as a result of a debate inside the thematic network Semantic Web Spain.

S. Omatu et al. (Eds.): IWANN 2009, Part II, LNCS 5518, pp. 91–98, 2009.

query and the expected result. OntologyTest allows the user to execute the tests, and to inspect the results of the execution. Thus, this tool can be considered a step towards dynamic testing similar to that used in software engineering [6].

Section 2 discusses the interest in performing functional tests for ontologies. Section 3 presents the features of the tool, and section 4 presents its internal architecture and its extensibility. A validation of OntologyTest is shown in Section 5. Finally, section 6 presents some conclusions on this work and future directions.

2 The Interest in Automating the Execution of Functional Tests

OntologyTest is a Java application that allows the elaboration and execution of tests to evaluate OWL DL ontologies. Such a tool can be seen as a complement of both Gruninger and Fox's methodology for the design and evaluation of ontologies [7], and of the NeOn European project approach of competency-question (CQ) driven ontology development [8]. According to this proposal, the activities to be carried out during the construction of an ontology are (i) to intuitively identify the main scenarios, that is, possible applications in which the ontology will be used; (ii) to obtain a set of natural language CQs; (iii) to use both these questions and their answers to extract the main concepts and their properties and relations; (iv) to formalize the CQs using the terminology obtained in the former activity; and (v) to write the formal axioms of the ontology.

Examples of competency questions are *what is the composition of paracetamol?* or *does paracetamol have interaction with another substance?* To obtain tests from a CQ, it must be formalized using SPARQL [9]. The first question expressed is SPARQL is shown in Figure 1.

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>
PREFIX chs:<http://www.owl-ontologies.com/subtances.owl#>

SELECT ?object
WHERE { chs:paracetamolHasPart rdfs:range ?object }
```

Fig. 1. Formalization of the competency question "what is the composition of paracetamol?" using the SPARQL query language

Natural language CQs are, on the one hand, the core of the functional specification of the ontology. On the other hand, they are a set of tests that the ontology must pass, although the ontological engineer may elaborate other tests for the ontology besides the formal CQs. For example, tests to check special attribute values.

OntologyTest allows the ontological engineer to capture the functional requirements of the ontology, both those that correspond to CQs, and those which do not, in a persistent and sharable representation. These requirements can be automatically checked at any time during the life cycle of the ontology. Therefore, it provides invaluable help to detect violations of the functional requirements, as well as regressions that may happen during the ontology's developments and evolution.

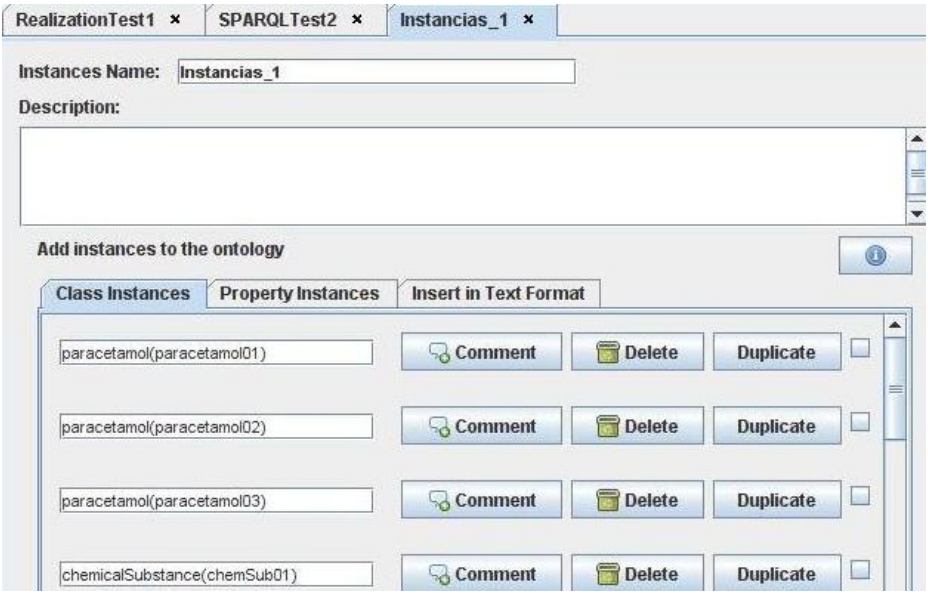

Fig. 2. OntologyTest screen where the different types of instances are created

Tools capable of capturing the functional requirements that an ontology must fulfil and automatically checking them, in general, OntologyTest, in particular, can potentially provide the ontological engineering field the same benefits that automated functional testing tools have provided to the software engineering field.

3 OntologyTest

To evaluate an OWL DL ontology using OntologyTest we must create a project, and indicate which ontology we are going to define a set of functional test for. The ontology must be saved locally and it must be consistent and syntactically correct. If it does not satisfy these properties, an error message warning the user about the problem is displayed when the ontology is loaded.

Then, the user can define the test queries that capture the functional requirements to be satisfied. The execution scenario of each test may comprise a set of instances that are not defined in the ontology and which we shall call *instance set* (see Fig. 2).

Each one of theses sets can be reused in different tests and can be copied, modified and associated with a different test. OntologyTest allows the user to create both class instantiations and property instantiations to make up an instance set. A class instantiation is a unary predicate saturated by a constant, for example, `paracetamol(paracetamol102)`. A property instantiation is a binary predicate saturated by two constants, for example, `hasInteractionWith(paracetamol102, ethylAlcoho01)`. To complete the definition of a test, the user must specify what the expected result is.

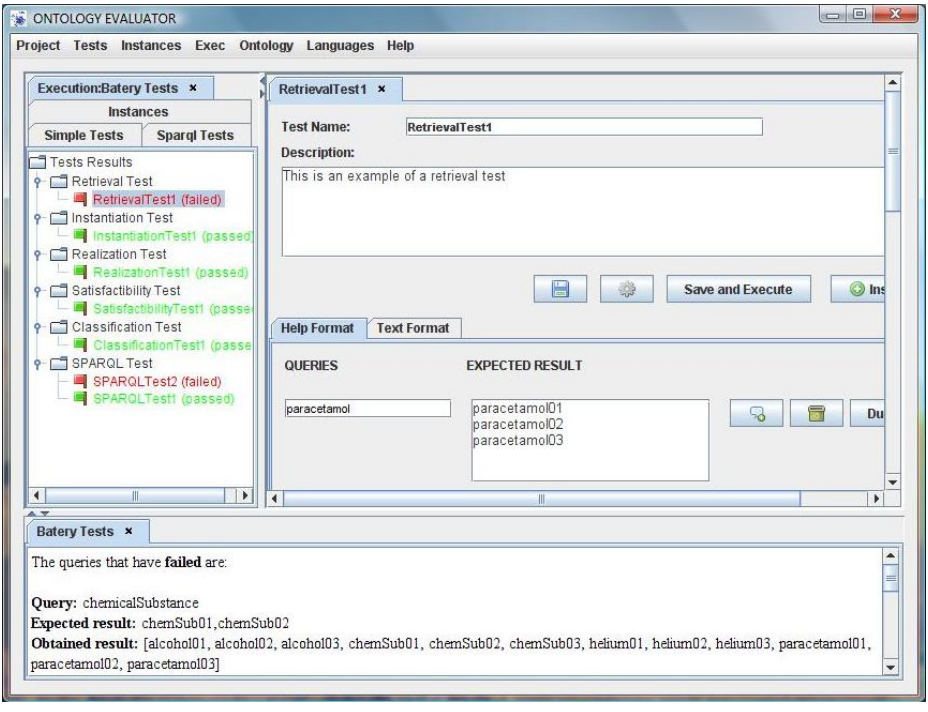

Fig. 3. Classification test with its queries, their expected results and the test's outcome

Sometimes this result is simply *true* or *false*. In other cases it may be a class or list of classes, or an instance or a list of instances.

Currently, the tool supports the following types of tests:

- *Instantiation test*. It specifies whether or not an individual belongs to a given class. Its queries follow the pattern *class(individual)*, and the possible answers are *true* –if it is expected for the individual to be an instance of the class– or *false* –otherwise. An example is *query*: `paracetamol(paracetamol102)`; expected result: `true`.
- *Recovering test*. It allows the user to specify a list with all instances that must belong to a particular class. Its queries follow the pattern *class*. The possible answer is the expected list of individuals that are members of the class. For instance, *query*: `paracetamol`; expected result: `[paracetamol101, paracetamol102, paracetamol103]`.
- *Realization test*. It specifies the most specific class that must be instantiated by an individual. Its queries follow the pattern *individual*. The possible answer is the class expected as the most specific that is instantiated by the individual. For instance, *query*: `paracetamol101`; expected result: `paracetamol`.
- *Satisfaction test*. It specifies whether an inconsistency should occur in the ontology after adding a new instance of a class. Its queries follow the pattern *class(individual)*. The possible answers are *true* –if it is expected that the

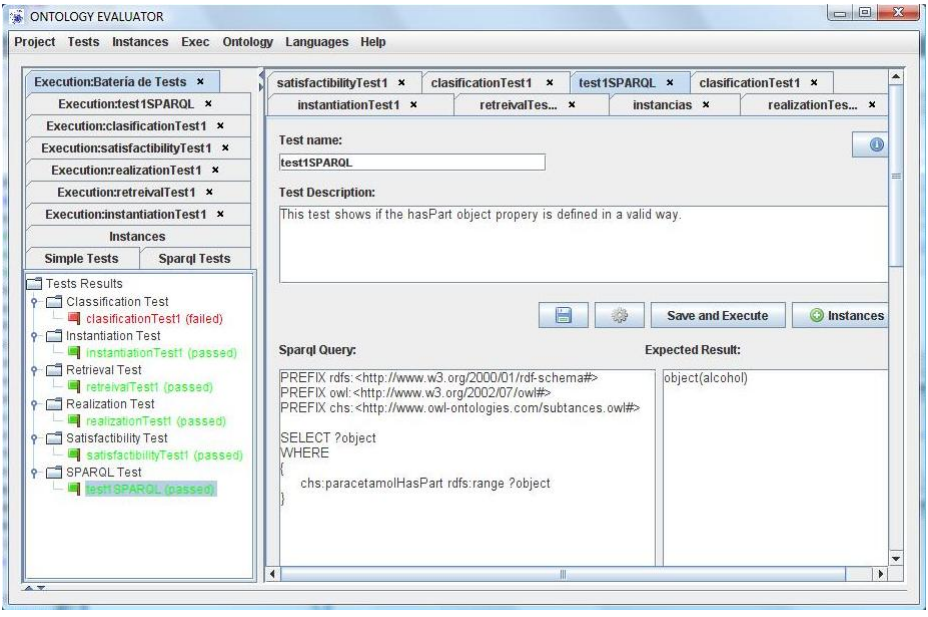

Fig. 4. SPARQL test query and its expected result

instantiation can be added without creating inconsistencies– or *false* –otherwise. An example is *query*: `paracetamol(paracetamol104)`; *expected result:* `true`.

- *Classification test*. It specifies a list with all classes that an individual must belong to. Its queries follow the pattern *individual*. The possible answer is the expected list of classes instantiated by the individual. For instance, *query*: `paracetamol101`; expected result: `[paracetamol, chemicalSubstance, thing]`. Figure 3 shows a classification test and its outcome in OntologyTest.
- *SPARQL test*. It is the most flexible and powerful type of test. The query is written in SPARQL, and the results are associated with the variables of the projection of the query. In Figure 1 a SPARQL test is shown; Figure 4 shows the same SPARQL test and its outcome in OntologyTest.

The definition of instance sets and the definition of the Instantiation, Recovering, Realization, Satisfaction, and Classification tests can be carried out with the support of a wizard – as shown in Figure 2– or by writing them in SPARQL. The first alternative requires less knowledge about the SPARQL syntax, while the second can be faster for experienced users. At any time, a test can be displayed in text format by clicking on the tab "Text Format" (see Figure 3). Any changes made in the text are automatically reflected in the wizard, and vice versa.

The test set is stored in an XML file; thus it is possible to share them among a group of engineers working on the same ontology, and they can be executed at any point during the life of the ontology.

Tests can be run individually, or a group of tests can be run at once. The results of their execution are shown using a tree where each node corresponds to a test; the ones which have passed are shown in green, and those that have failed are shown in red. By clicking on the node corresponding with a failed test an explanation about why it failed can be obtained. More detailed information about the execution of the tests can also be obtained by consulting a text field at the bottom of the tool that acts as a console (see Figure 3).

4 Architecture and Extensibility

The internal architecture of OntologyTest is inspired by the architecture of the popular Java unit testing framework JUnit [10] which in turn –as all the other xUnit frameworks–, is based on the SUnit Smalltalk testing framework, developed by Ken Beck in 1998.

The execution of every test is carried out in three well defined steps: (1) preparation of the initial conditions for the test, which often requires the creation of a set of instances in the ontology, (2) the execution of the test; (3) and rolling back the ontology to its initial state to prevent the execution of one test from affecting the outcome of the following tests. The Template Method design pattern [11] is used to enable the redefinition of any of these steps, while still permitting a uniform management of the different types of tests –instantiation, recovery, realization, classification, satisfiability, and SPARQL test– supported in the tool. A superclass – *OntologyTestCase* – defines three methods that act as placeholders for the specific operations to be carried out in each of these three steps –*setUpOntology*, *runOntologyTest* and *tearDownOntology*. The subclasses of this class provide concrete implementations for the three steps. Therefore, it is possible to create new subclasses that redefine one or more of the steps of a test execution without losing the ability to manage all of the subclasses uniformly through the superclass interface.

One of the key differences with the xUnit framework family is that the tests that are executed over an ontology are declarative, and not procedural. This has allowed us to create a graphical interface that enables tests to be defined without the need for writing code in some programming language. This is of paramount importance because, unlike the software engineer who uses an xUnit framework, the ontological engineer that wants to test an ontology does not necessarily need to be familiar with the programming language in which the testing framework has been built – as in the case of OntologyTest, Java.

Another difference with respect to the xUnit framework family is the necessity of relying on a reasoner for carrying out the tests. If OntologyTest had imposed the use of a single reasoner, the usefulness of the tool would be limited to the cases in which this reasoner could be utilized. This has led us to completely isolate the representation and persistence of the test, as well as the execution engine of the tests from specific reasoners. We have defined an interface –*ReasonerInterface*– that contains all the operations that must be supported by a reasoner in order to be employed by OntologyTest, as well as an exception that should be used to encapsulate any reasoner specific exceptions that may be thrown when querying or updating an ontology. The concrete class that implements *ReasonerInterface* and that allows OntologyTest to use

a specific reasoner for the test execution is loaded dynamically using the reflection capabilities of the Java programming language. The name of this class is specified in a configuration file. This solution is inspired by the Java JDBC API.

Currently, OntologyTest has a driver that uses the open-source Java based OWL DL Reasoner Pellet [12], along with Jena. Thus, it is limited to working with OWL DL ontologies.

5 Validation

OntologyTest has been evaluated through (meta-)tests. Each functional requirement of the tool –e.g. create, modify and delete test– has been evaluated, at least, by means of a medium-sized and a large ontology. OntologyTest has also undergone numerous tests checking the correct functioning of all types of tests supported by the tool. Each type of test was assessed several times with various ontologies of different sizes.

The tool efficiency has also been tested under stress conditions: ontologies containing a large number of definitions have been loaded on the tool, and queries causing a large amount of data to be returned were made. Some of the large ontologies have been automatically generated by a Java program. Other ontologies have been taken from the Internet. For example, UMLS [13], an ontology of 135 concepts and 133 axioms is loaded and its tests executed virtually immediately by an Intel Pentium to 3 GHz and 992 MB of RAM. The Ontosem ontology [14], which has 7596 concepts, 604 object properties and 7992 axioms, is loaded in 38 seconds. The execution of a test involving a query that retrieved all of the 7596 concepts took approximately 30 seconds.

6 Conclusions and Future Directions

Until now, neither a method nor a tool for the elaboration of dynamic tests to check an ontology with regard to its functional requirements had been proposed. OntologyTest has been developed to overcome this drawback. This tool allows the user to define a set of tests and to run them automatically. Each one of them comprises an initial scenario, a query or set of queries, and the expected result. Their outcomes are shown in a style similar to the one used in XUnit software testing tools. The overall appearance of OntologyTest is similar to that of any Integrated Development Environment. This tool can be considered a technological contribution to CQ-driven ontology development. Besides, it provides invaluable aid in detecting violations of the functional requirements of an ontology, as well as regressions that may occur during the ontology's development and evolution.

Regarding OntologyTest validation, although it has been evaluated through a series of (meta-)tests, we intend to carry out a beta testing in a community of users (e.g. Geobuddies project, Neon, etc.).

Some possible extensions of OntologyTest are the automatic monitoring of the resource consumption of each query and the capability of testing ontologies that are not saved locally.

It is also interesting to study the impact of this tool in current methodologies, using software engineering as a parallel field. For example, once we have test automation, Can agile development be also applied to ontology building? Our previous experience indicates that an agile approach can be beneficial when a large effort to make explicit the knowledge is not necessary. In the opposite case, this approach may not be the most appropriate. However, ontologies represent concepts agreed upon by a community, therefore, the first scenario is more common.

Acknowledgment

This work was supported by the Spanish MEC and the European FEDER under the grants TSI2007-65677-C02-01 and TIN2006-15460-C04-02; by the Xunta de Galicia under the grant 08SIN002206PR; and by the University San Pablo CEU under the grant USP-PPC 04/07.

References

1. Gruber, T.R.: A translation approach to portable ontology specification. Knowledge Acquisition 5(2), 199–220 (1993)
2. Dean, M., Schreiber, G.: OWL Web Ontology Language Reference. W3C Recommendation (2004), http://www.w3.org/TR/owl-ref/
3. Gómez-Pérez, A., Fernández-López, M., Corcho, O.: Ontological engineering. Springer, London (2003)
4. Ontology Evaluation. Buffalo Ontology site, http://ontology.buffalo.edu/evaulation.html
5. Hartmann, J., Spyns, P., Giboin, A., Maynard, D., Cuel, R., Suárez-Figueroa, M.C., Sure, Y.: Methods for ontology evaluation. Knowledgeweb European Project, D.1.2.3 deliverable (2005)
6. Perry, W.: Effective methods for software testing. John Wiley & Sons, New York (1995)
7. Grüninger, M., Fox, M.S.: Methodology for the design and evaluation of ontologies. In: Skuce, D. (ed.) IJCAI 1995 Workshop on Basic Ontological Issues in Knowledge Sharing, pp. 6.1–6.10 (1995)
8. Suárez-Figueroa, M.C.(coord.): NeOn Methodology for Building Contextualized Ontology Networks. NeOn European Project Deliverable 5.4.1 (2008)
9. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (2008), http://www.w3.org/TR/rdf-sparql-query/
10. Gamma, E., Beck, K.: JUnit, http://www.junit.org
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Reading (1995)
12. Pellet: The Open Source OWL DL Reasoner, http://clarkparsia.com/pellet
13. Unified Medical Language System, http://www.nlm.nih.gov/research/umls/
14. Ontosem ontology, http://morpheus.cs.umbc.edu/aks1/ontosem.owl