Information Sciences 582 (2022) 89–113

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Ontology verification testing using lexico-syntactic patterns

Alba Fernández-Izquierdo *, Raul García-Castro

Ontology Engineering Group, Universidad Politécnica de Madrid, Spain

ARTICLE INFO

Article history:
Received 5 March 2021
Received in revised form 23 August 2021
Accepted 5 September 2021
Available online 8 September 2021

Keywords:
Ontology testing
Ontology verification
Ontology requirements

ABSTRACT

Ontology verification refers to the activity where an ontology is tested against its ontology requirements to ensure that it is built correctly in compliance with its ontology requirements specification. Therefore, it is an important activity that should be performed in any ontology development process. Since manual verification can be a time-consuming and repetitive task, testing processes to automatically verify an ontology facilitate this activity. Moreover, the involvement of not only ontology engineers during the ontology verification process, but also domain experts and users, can provide valuable feedback to avoid misunderstandings and lack of information. This paper proposes a method for ontology verification that defines the testing activities to be performed. The method uses a testing language based on lexico-syntactic patterns to facilitate the definition of tests and an ontology to store and publish such tests. Moreover, this verification testing method proposes an online tool to execute tests on one or more ontologies. The method was compared in terms of time and errors by user evaluation with other tools for ontology verification; the evaluation showed that the tools that use testing languages had better results in terms of reducing errors in the verification activity compared to the tools that do not.

1. Introduction

Ontology verification refers to the ontology evaluation activity where the ontology is compared against its ontology requirements, ensuring that the ontology is built correctly in compliance with the ontology requirements specification [30].

Ontology Engineering has been inspired over the years by Software Engineering practices but, regarding ontology verification, there is still much to learn from software verification approaches. In this field, software verification practices and techniques are widely integrated into the software development process to ensure the quality of software products. However, in the Ontology Engineering field, ontology verification has been neglected, and only a few approaches deal with this activity (e.g., Blomqvist and colleagues' testing methodology [6] or the test-driven development of ontologies proposed by Keet and Ławrynowicz [20]). Therefore, the adoption of software practices and techniques related to verification can be beneficial for Ontology Engineering.

Inspired by the Software Engineering field and since manual ontology verification can be a time-consuming and repetitive task, the latest works in Ontology Engineering propose the use of tests to automate and facilitate the ontology verification [28]. However, the creation of such tests is not a trivial task, especially if tests are written in a formal language such as SPARQL. This is one of the reasons for the rise of testing syntax in the Software Engineering field, such as Gherkin,[1] which

* Corresponding author at: Ontology Engineering Group, Universidad Politécnica de Madrid, Spain.
E-mail address: albafernandez@fi.upm.es (A. Fernández-Izquierdo).

[1] https://cucumber.io/docs/gherkin/.

https://doi.org/10.1016/j.ins.2021.09.011
0020-0255/

emerged intending to describe testing behaviours without the need to go into implementation details. Through these testing behaviours, tests can be easily understood by experts and non-experts, and they can be automatically executed on a software product.

During the ontology verification testing activity, in addition to ontology engineers, domain experts, and users can also be involved. Feedback between them and ontology engineers can help identify misunderstandings in requirements or lack of information at an early stage of the development process. Moreover, involving users would increase their confidence in the correctness of an ontology, since they would be able to check and understand the testing results and would support knowledge capture. Therefore, a verification testing process should define a particular testing language that facilitates the definition of requirements-based tests, allowing developers, domain experts, and users to participate in the testing process.

This paper presents an ontology testing method for ontology verification, which defines a novel testing language based on how ontology requirements are specified using lexico-syntactic patterns [26] to facilitate the generation of tests and reduce errors made by users in the identification of verified requirements. Moreover, it defines the set of testing activities to be carried out to systematise the verification process. These activities are divided into test design, test implementation, and test execution, separating the test design from its implementation to increase the maintenance of the tests, since the test implementation can be updated without changing the test design. Moreover, this separation between test design and test implementation allows using a specific syntax in test design to ease the definition of tests. This method also enables to maintain the traceability between requirements, tests, and ontology implementation by proposing an ontology to store these artefacts in the RDF format.

This ontology verification testing method was used as the basis for the conformance testing method proposed by Fernández-Izquierdo and García-Castro [13], in which the tests are used to check whether an ontology satisfies the requirements of a standardized document, e.g., standard ontologies or ISO documents.

With the ontology verification testing method proposed in this paper, this work aims at addressing the following research questions to analyse how to ease the verification process for ontology engineers, domain experts, and users:

RQ1. *What is the reduction in terms of errors in the identification of verified requirements for those verification methods that use a testing language compared to those that do not?*
RQ2. *What is the reduction in terms of reply time during the verification activity for those verification methods that use a testing language compared to those that do not?*
RQ3. *Are users more satisfied regarding usability when using a verification method that uses a testing language?*

Based on such three research questions, the following hypotheses are made:

H1. *Using a testing language to define tests based on functional requirements facilitates the ontology testing process regarding the reduction of time in users who are familiar and experienced in OWL.*
H2. *Using a testing language to define tests based on functional requirements facilitates the ontology testing process regarding the reduction of errors in users that are familiar and experienced in OWL.*
H3. *Using a testing language to define tests based on functional requirements increases the usability of the tools during the verification activity.*

Consequently, the contributions of this paper are the following:

C1. A verification testing method that uses lexico-syntactic patterns and that systematises the verification activity. The method considers the participation of ontology engineers, domain experts, and users during the verification activity. Moreover, the method stores the requirements, tests and their results in a machine-readable format, enabling traceability between the different ontology artefacts involved in the verification activity.
C2. A testing language that expresses its syntax using lexico-syntactic patterns to facilitate the definition of tests for different types of users. Moreover, to generate the testing language, representative keywords based on ontology axioms have been extracted from each lexico-syntactic pattern to propose the set of test designs.
C3. A comparison between verification tools that use testing languages with tools that do not. This comparison allows analysing whether using a testing language reduces errors in the identification of verified requirements. Moreover, it also allows analysing whether using testing languages reduces the reply time during the verification activity.

The remainder of this paper is structured as follows. Section 2 presents the state of the art related to existing verification testing methods. Section 3 describes the proposed method for verification testing, together with the developed ontology for promoting traceability and technological support. Section 4 describes the validation of the proposed research questions and hypotheses. Finally, Section 5 outlines some conclusions and future lines of work.

2. State of the art

Several approaches which defend the importance of verifying ontologies through their ontology requirements have been developed to date. These approaches focus on some aspects: requirements analysis, verification testing methodology, query execution, and integration of ontology testing into ontology development methodologies. This section summarises the existing approaches that can be used to verify whether particular functional requirements are satisfied in an ontology. It should be noted that these methods are oriented to the verification of OWL ontologies.

2.1. Requirements analysis

Several works are focused on the analysis of how requirements are specified, with the ultimate goal of helping their translation into formal queries that can be executed on an ontology. These works are not focused on providing a process for automatically translating natural language requirements into formal queries, but on providing lists of linguistic patterns that can be used to make such translations.

One of these works is the approach presented by Ren and colleagues [29], which focuses on authoring driven by competency questions. In this work, the authors use natural language processing and patterns to analyse requirements written in the form of competency questions with the ultimate goal of automatically obtaining SPARQL queries to test an ontology. Therefore, this analysis of requirements can be used for the definition of tests to verify an ontology. The authors analysed a set of 75 competency questions to identify the patterns to which they belong based on several features. Examples of these patterns are *Question type*, which determines the kind of answers presented when answering the competency question, such as selection questions, binary questions and counting questions, and *Element Visibility*, which indicates whether the modelling elements, such as class expressions and property expressions are explicit or implicit in the competency question.

Based on these features, this work categorises competency questions into archetypes, which are shown in Table 1.

Another work oriented to requirements analysis is the one proposed by Wiśniewski and colleagues [34], which focuses on the association between linguistic patterns and SPARQL-OWL queries [22]. The authors analysed 234 requirements written as competency questions that are associated with 5 different ontologies and extracted 81 distinctive linguistic patterns. This set of linguistic patterns extended the one proposed by Ren and colleagues, which was based on a smaller dataset of requirements written as competency questions.

To perform the analysis, the authors proposed a method that includes the following steps:

1. *Chunks and pattern candidates.* In order to identify regularities among the collected questions, the authors studied the linguistic structure of every requirement in the dataset. Because the requirement dataset does not contain any pair of questions consisting of identical sequences of words, they proposed a pattern detection procedure to identify more general groups.
2. *Pattern semantics.* The previous steps produced some patterns that are semantically the same but that differ in minor aspects such as using plural and singular verbs, using synonyms or using words that could be removed from the requirement without changing its meaning. The authors semantically joined the same patterns and unify all cases, e.g., *Is there CE1 for CE2?* and *Are there any CE1 for CE2?* represent the same pattern.

In this work, the authors concluded that there can be multiple SPARQL-OWL queries for one distinct linguistic pattern, which is due to the different ways an ontology engineer can represent knowledge in the ontology. Moreover, there can be multiple linguistic patterns for a single SPARQL-OWL query, because there are different ways to formulate the same thing in natural language.

2.2. Ontology verification testing methods

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

Blomqvist and colleagues [6] present an agile approach which includes a methodological background for testing and introduces three main types of tests, namely: (1) competency question verification, (2) inference verification, and (3) error provocation. The first type of test is oriented to the reformulation of the competency questions as SPARQL queries after adding test data related to the query to be reformulated, i.e., it does check the existence of classes and properties in the ontology. The second type verifies that the inference mechanisms are in place. Finally, the third is intended to expose faults. To keep the tests separated from the ontology to be tested, this proposal represents a test case as an OWL ontology, which includes properties for describing each test case.

Table 1
Competency question archetypes1 [29].

ID	Pattern	Example
1	Which [CE1] [OPE] [CE2]?	Which pizzas contain pork?
2	How much does [CE] [DP]?	How much does Margherita Pizza weigh?
3	What type of [CE] is [I]?	What type of software (API, Desktop application etc.) is it?
4	Is the [CE1] [CE2]?	Is the software open source development?
5	What [CE] has the [NM] [DP]?	What pizza has the lowest price?
6	What is the [NM] [CE1] to [OPE] [CE2]?	What is the best/fastest/most robust software to read/edit this data?
7	Where do I [OPE] [CE]?	Where do I get updates?
8	Which are [CE]?	Which are gluten free bases?
9	When did/was [CE] [PE]?	When was the 1.0 version released?
10	What [CE1] do I need to [OPE][CE2]?	What hardware do I need to run this software?
11	Which [CE1] [OPE] [QM] [CE2]?	Which pizza has the most toppings?
12	Do [CE1] have [QM] values of [DP]?	Do pizzas have different values of size?

[1] In the table, CE = class expression, OPE = object property expression, DP = datatype property, I = individual, NM = numeric modifier, PE = property expression, QM = quantity modifier.

Although this methodology proposes types of tests to verify requirements, it does not describe the relation between such requirements and the tests, e.g., when to use each type of test. Moreover, the analysis of the requirements is out of the scope of this methodology.

CQChecker[2] [5] is a tool proposed by Bezerra and colleagues which provides a mechanism that allows verifying whether the ontology meets its corresponding competency questions by supporting both assertional and terminological queries. To accomplish that, the authors distinguish several types of competency questions, such as competency questions which work over classes and their relations or competency questions which work over instances. They create a modular architecture in which each module treats each different competency question.

The authors identify three types of competency questions based on how they are specified:

1. Competency questions that work with classes and their relations.
2. Decision problems expressed as competency questions. In this type, the answer permitted to the question can only be true or false.
3. Competency questions expressed in an interrogative form that works only over instances.

The tool analyses the competency question submitted by the user to classify it into one of the three types of competency questions according to the possible answer it is supposed to retrieve. Then, the system processes it and checks whether it is satisfied.

2.3. Ontology testing methods for query execution

While in the previous sections several testing approaches and tools based on competency questions have been reviewed, this section focuses on tools that query an ontology and that can be used for ontology verification.

These approaches are not oriented to the definition of types of tests based on different types of requirements, but to the definition of different types of queries that can be executed on an ontology. Although these approaches are not directly oriented to the ontology verification activity, they can be useful for analysing how they support the execution of queries to test an ontology.

OntologyTest[3] [17] is a Java application that allows the elaboration and execution of tests to evaluate OWL DL ontologies. Each test comprises an optional set of instances, a query, and the expected result. OntologyTest supports the following types of tests:

- *Instantiation test*. It specifies whether or not an individual belongs to a given class. An example of query could be *Paracetamol(paracetamol102)*, which checks whether the instance *paracetamol102* is of class *Paracetamol*.
- *Recovery test*. It allows the user to specify a list with all instances that must belong to a particular class. An example of query could be *Paracetamol*, and an example of expected result could be *[paracetamol101, paracetamol102, paracetamol103]*, which represent all the instances of the *Paracetamol* class.
- *Realisation test*. It specifies the most specific class that must be instantiated by an individual. For instance, the query *paracetamol101* and the expected result *Paracetamol*, check whether *paracetamol101* is an instance of the class *Paracetamol*.

[2] CQChecker is not available online.
[3] OntologyTest is not available online.

- *Satisfaction test*. It specifies whether an inconsistency should occur in the ontology after adding a new instance of a class. An example of query could be *Paracetamol(paracetamol102)*, to check whether the addition of the instance *paracetamol102* leads to an inconsistency.
- *Classification test*. It specifies a list with all the classes that an individual must belong to. For instance, the query *paracetamol101* with the expected result *[paracetamol, chemicalSubstance, thing]* indicates all the classes to which the instance *paracetamol101* belongs to.
- *SPARQL test*. The query is written in SPARQL, and the results are associated with the variables of the query.

VoCol.[4] Ref. [19] is an integrated environment that supports the development of vocabularies using version control systems. VoCol supports a round-trip model of vocabulary development, consisting of three core activities, i.e., modelling, population, and testing. For testing, VoCol allows formulating SPARQL queries that represent competency questions.

Although VoCol provides a service that allows users to execute SPARQL queries on an ontology, how to create such queries is out of scope of the tool.

2.4. Integrated ontology testing into ontology development methodologies

Keet and Ławrynowicz proposed a test-driven development (TDD) method for ontologies [20], which is an ontology development approach inspired by test-driven development in Software Engineering [3]. This TDD, which is based on the idea of writing a failing test before writing any code, ensures that what is added to the program core does indeed have the intended effect specified upfront. Moreover, the test-driven development principle increases the understanding of the ontology authoring process and the logical consequences of an axiom.

The steps to be performed within the TDD testing approach are summarised as follows:

1. To check whether the vocabulary elements of the axiom x are in the ontology O (itself a TDD test).
2. To run the TDD test twice:
   (a) To run the first execution of the tests. This first execution should fail.
   (b) To update the ontology (add x).
   (c) To run the test again, which should pass, checking that there is no new inconsistency or undesirable deduction.
3. To run all previous successful tests, which should pass. This step represents the regression testing.

This TDD methodology is supported by the TDDOnto tool, which has been first introduced by Ławrynowicz and Keet [23] and further developed as TDDOnto2 [9].[5] This tool checks whether a particular axiom is present in an ontology. TDDOnto2 is based on a logic-based model of TDD unit testing and on generalised versions of the TDD algorithms [20] to support tests related to any class expression of OWL 2.

2.5. Conclusions

Based on the state of the art analysis on ontology verification, it can be concluded that in the current testing approaches and tools there is a lack of guidelines to help developers and practitioners to create tests from such requirements.

Moreover, the majority of these testing approaches do not consider traceability between requirements, tests, and ontologies, and are oriented to ontology engineers, e.g., developing plugins for ontology engineers tools or generating SPARQL queries. Therefore, they pay little attention to how both domain experts and users should also become part of the ontology verification activity.

Table 2 summarises the conclusions obtained from the analysis of the testing approaches in Ontology Engineering, indicating how the methods and tools support traceability, definition, implementation, and execution of tests, and which is the targeted audience for each.

The method proposed in this paper aims at addressing these current lacks by providing a testing method that uses a testing language based on how ontology requirements are specified using lexico-syntactic patterns. Such a testing language facilitates the definition of tests not only for ontology engineers but also for users and domain experts. Moreover, the method proposes the storage of the results of the verification testing process in an RDF file, which enables traceability between the tests, requirements, and the ontology.

3. A method for ontology verification testing

The method for ontology verification testing proposed in this paper aims to involve different roles in the ontology verification activity in addition to ontology engineers, i.e., domain experts and users. These roles can be defined as follows:

[4] VoCol is available in the following URL: https://www.vocoreg.com/. The last update was on 6 April 2021.
[5] The last version of TDDOnto is available at the following URL: https://github.com/kierendavies/tddonto2. The last update was on August 23, 2018.

Table 2
Summary of conclusions regarding ontology testing approaches.

	Ren and colleagues	Wiśniewski and colleagues	Blomqvist and colleagues	CQChecker	OntologyTest	VoCol	TDD
			Dimension				
Traceability	Not treated	Not treated	Test stored in an ontology	Not treated	Not treated	Not treated	Not treated
			Detailed Guidelines for Processes and Activities				
Specification of requirements	Patterns of competency questions	Patterns of competency questions	Competency questions	Competency questions	Not provided	Not provided	Not provided
Definition of tests of competency questions	List of types List of types of competency questions	List of types of tests	Not provided	Not provided	Not provided	Tests as OWL axioms	
Implementa- tion and execution of tests	SPARQL queries and inference checking	SPARQL queries	SPARQL queries	SPARQL queries	SPARQL queries	SPARQL queries	Presence or absence of axioms
			Audience				
Target audience	Ontology engineers	Ontology engineers	Ontology engineers	Ontology engineers	Ontology engineers	Ontology engineers	Ontology engineers
			Technological support				
Tool	No	No	Yes	Yes	Yes	Yes	Yes

- Ontology engineer: An ontology engineer is a member of the ontology development team who has high knowledge about ontology development and knowledge representation. The ontology engineer is usually the person who defines and executes the tests.
- Domain expert: A domain expert is an expert in the domains covered by the ontology. This role does not need to be knowledgeable about ontology development, but a domain expert can use the verification testing method to check whether the ontology satisfies the domain that is expected to be covered.
- Ontology user: An ontology user is a potential end user of the ontology. This actor also includes software developers who will use the ontology in their applications. This role can also use the verification testing method to check whether the ontology satisfies their needs.

In this method, although usually the ontology engineer is the person who defines the set of tests, the domain experts and users could also participate in it and use the method. To this end, it is necessary to use a language that is understandable by all of these roles. Therefore, inspired by software testing approaches such as keyword-driven testing [33] or behaviour-driven development [24], which are widely adopted in software development, this testing method defines a testing language based on how the requirements are specified.

Furthermore, this testing method aims at systematising the design, implementation, and execution of tests extracted from ontology requirements to verify an ontology, as well as at providing traceability between ontology requirements, tests, and ontologies. Consequently, this method describes the set of activities to be carried out in a verification testing process and proposes an ontology to store and publish the tests. This section provides details of the proposed ontology verification testing method.

Following agile and iterative methodologies, such as LOT,[6] this verification activity should be performed once the ontology is modified, leading to the creation of a new ontology version or a new ontology sprint. This process would also allow to perform regression testing and check whether the requirements are still satisfied over time after the ontology changes.

3.1. Ontology requirements specification analysis

In recent years, the definition of ontology requirements [4,31,32], which represent the needs that the ontology to be built should cover, and their automatic formalisation into axioms or tests [10,29,35] have been studied. These studies aim to reduce the time consumed by ontology engineers during the ontology verification activity. Ontology requirements can be written in the form of competency questions [18], which are natural language questions that the ontology to be modelled should be able to answer, or as statements.

However, accurately defining ontology requirements is not a trivial task and, therefore, neither is their automatic translation into a formal language. Due to the fact that some requirements are ambiguous [26] or vague, their transformation into axioms or tests is not usually direct [10] and, consequently, it is challenging to automate such translation. Therefore, this

[6] https://lot.linkeddata.es/.

method benefits from the use of Lexico-Syntactic Patterns (LSPs), which represent linguistic schemas or constructions derived from regular expressions in natural language. These LSPs consist of certain linguistic and para-linguistic elements that allow one to extract conclusions about the meaning expressed by the construction [1]. Therefore, they are used to analyse how requirements are specified, with the ultimate goal of identifying the types of tests based on them.

CORAL (Corpus of Ontology Requirements Annotated with Lexico-syntactic patterns) [14] is a corpus of 834 functional ontology requirements collected from different projects, websites, and papers. It can be used as a resource to help the formalisation of ontology requirements in an ontology since it provides a dictionary of LSPs, which includes those LSPs collected from the state of the art and those new ones identified based on the requirements gathered in CORAL. Moreover, it provides a potential implementation of each LSP in an ontology. This dictionary of LSPs has been used to annotate the set of 834 ontology requirements in order to determine the LSPs that these requirements follow. Furthermore, these LSPs identify structures in the specification of ontology requirements; consequently, each LSP represents a particular type of requirement. CORAL is openly available in HTML,[7] CSV, and RDF formats as Zenodo resources.[8]

In the CORAL dictionary of LSPs, an LSP may be associated with several disjoint ODPs, resulting in several possible ontology implementations and, therefore, needing a different group of OWL constructs for each one. The translation from these polysemous LSPs to a formal implementation is not direct. Consequently, in such cases, ontology engineers should decide which possible implementation they prefer according to the ontology they are implementing.

In this dictionary, the LSPs are categorised according to their correspondences, which could also be used to identify whether the LSP patterns are polysemous, i.e., *1 to 1* correspondence if one LSP corresponds to one possible implementation in the ontology or *1 to N* correspondence if one LSP can result in more than one possible implementation in the ontology. The LSPs that have more than one correspondence are considered polysemous LSPs.

Table 3 summarises the LSPs according to their correspondence with ontology axioms. This table also includes the sources from which each LSP has been extracted since some of them were defined in previous works [8].

3.2. Activities within the ontology testing method

This section details the activities to be carried out during the testing method for verifying an ontology, which were first introduced in [12]. Since then, these activities have evolved to integrate the testing language proposed in this paper. In the literature, ontology testing approaches are usually divided into two activities, i.e., *test implementation* and *test execution*. However, based on the requirements specification analysis described in Section 3.1, in this testing method a new activity is proposed, i.e., *test design*. This new activity is needed due to the ambiguity and assumptions inherent in natural language [10] and to the fact that different people may be in charge of the design and implementation of tests. Additionally, the separation between test design and test implementation increases the maintainability of the tests, since the implementation can change without updating the design of the test, and allows the test implementation to be generated automatically from the test design. Therefore, in this design activity, the goal of each requirement is identified and specified using a testing language based on the LSPs presented in Section 3.1.

The following sections present each of these testing activities, which are also summarised in Fig. 1. The *test design* activity is a manual activity that should be performed by ontology engineers and practitioners, while the *test implementation* and the *test execution* activities can be carried out automatically.

3.2.1. Test design activity

Ontology engineers, domain experts, and users should be involved in the ontology verification testing process, where tests are generated from the requirements in order to ensure that the ontology satisfies all expected needs. However, writing tests in formal languages such as SPARQL, which is a language whose semantics are not easy to understand for people without a background on it [27], is not a trivial process.

In Software Engineering, there are several techniques and languages to ease the test design process. An example is keyword-driven testing [33], which aims to express each test as abstractly as possible, but still precise enough to be executed and interpreted by a test execution tool. Another example is behaviour-driven development [24], which proposes a common language for specifying system behaviours. Inspired by them, this method describes a testing language to design tests. Since the LSPs introduced in Section 3.1 determine the types of requirements and how they should be implemented in an ontology, the testing language is grounded on them.

To that end, representative keywords have been extracted from each LSP to define the syntax of the test designs. These keywords are based on the ontology axioms (e.g., *rdfs:subClassOf*) and on the different types of LSP [14] (e.g., hierarchies between terms). Therefore, these keywords indicate the goal in the ontology for each LSP, such as to define a subsumption relation or a cardinality restriction, and are used to propose a catalogue of test expressions that are written following the OWL Manchester syntax.[9] Therefore, each test refers to a particular type of requirement and includes a set of keywords, e.g., the tests related to subsumption relations between classes includes the keyword *subClassOf*. Since these tests are written using keywords, they can be automatically analysed and implemented into queries or axioms to be executed on the ontology,

[7] http://coralcorpus.linkeddata.es.
[8] https://doi.org/10.5281/zenodo.1967306.
[9] https://www.w3.org/TR/owl2-manchester-syntax.

Table 3
List of lexico-syntactic patterns included in the CORAL dictionary [14].

LSP	Type of correspondence
	1 to 1 correspondence
LSP-SC-EN	LSP for subclassOf relation ODP
LSP-MI-EN	LSP for multiple inheritance ODP
LSP-EQ-EN	LSP for equivalence relation ODP
LSP-OP-EN	LSP for object property ODP
LSP-DP-EN	LSP for datatype property ODP
LSP-Di-EN	LSP for disjoint classes ODP
LSP-SV-EN	LSP for specified values ODP
LSP-PA-EN	LSP for participation ODP
LSP-PCP-EN	LSP for co-participation ODP
LSP-LO-EN	LSP for location ODP
LSP-OR-EN	LSP for object-role ODP
LSP-DC-SC-EN	LSP for defined classes and subclass
LSP- SC-Di-EN	LSP for subclass relation, disjoint classes and exhaustive classes
LSP-OP-UR-EN	LSP for object property and universal restriction
LSP-CD-EN	LSP for class definition
LSP-Min-and-OP-EN	LSP for object property minimum cardinality and object property
LSP-OP-Min-EN	LSP for object property minimum cardinality related to an object property
LSP-OP-Max-EN	LSP for object property maximum cardinality related to an object property
LSP-OP-Exact-EN	LSP for object property exact cardinality related to an object property
LSP-SYM-EN	LSP for symmetry
LSP-U-EN	LSP for union
LSP-INTER-EN	LSP for intersection
LSP-COMPL-EN	LSP for complement
	1 to N correspondences
LSP-SC-PW-EN	LSP for subclass or simple part-whole relation
LSP-OP-DP-PW-EN	LSP for object property or datatype property or simple part-whole relation
LSP-PW-CONS-COM-CE-EN	LSP for simple part-whole relation or constituency or componency or collection-entity
LSP-DC-SC-EN	LSP for defined classes and subclass relation
LSP-INST-SC-EN	LSP for instances of subclass relation
LSP-OP-DP-EN	LSP for object property or datatype property

enabling test automation. The person that is in charge of defining the tests is the one responsible for determining which test design from the list fits each ontology requirement.

Due to the fact that polysemous LSPs do not have a direct translation from requirements into an ontology, since multiple ontology implementations can be correct, they are not considered for the time being. Additionally, it is worth mentioning that a small set of tests and keywords was also added to the catalogue on the demand of ontology experts.

The catalogue of test expressions is shown in Table 4. The table includes the goal of each test, its syntax, and its provenance. Those terms that are represented between brackets (e.g., "[Class]") correspond to those terms that should be completed by the user, while those terms italicised (e.g., *type*) correspond to the fixed keywords extracted from the LSPs that cannot be changed. Table 4 is divided into three categories, according to the axioms analysed by each test:

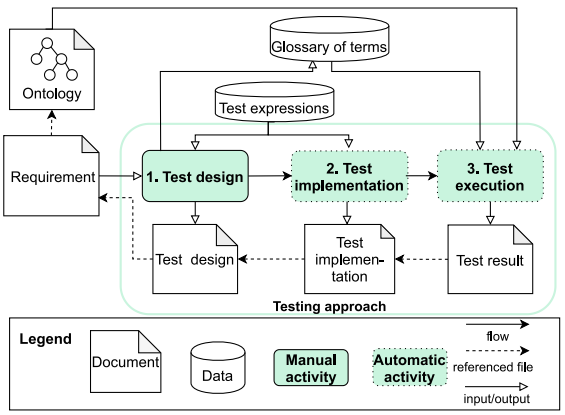


Fig. 1. Testing activities during the testing process, together with their inputs and outputs.

Table 4
List of proposed test expressions together with their category.

Goal	Syntax	Source
	Simple test	
T1. Class A exists	[ClassA] *type Class*	LSP-CD-EN
T2. Subsumption relation between classes A and B	[ClassA] *subclassOf* [ClassB]	LSP-SC-EN
T3. Disjointness between two classes	[ClassA] *disjointWith* [ClassB]	LSP-Di-EN
T4. Equivalence between two classes	[ClassA] *equivalentTo* [ClassB]	LSP-EQ-EN
T5. Property P exists	[Property] *type Property*	Ontology experts
T6. Existential relation P between two classes A and B	[ClassA] *subclassOf* [PropertyP] *some* [ClassB]	LSP-OP-EN, LSP-DP-EN, LSP-LO-EN, LSP-SV-EN
T7. Universal relation P between two classes A and B	[ClassA] *subclassOf* [PropertyP] *only* [ClassB]	LSP-OP-EN, LSP-DP-EN, LSP-LO-EN, LSP-SV-EN
T8. Symmetric property P	[PropertyA] *characteristic symmetric*	LSP-SYM-EN
T9. Minimum cardinality	[ClassA] *subclassOf* [PropertyP] *min* [num] [ClassB]	LSP-OP-Min-EN
T10. Maximum cardinality	[ClassA] *subclassOf* [PropertyP] *max* [num] [ClassB]	LSP-OP-Max-EN
T11. Cardinality	[ClassA] *subclassOf* [PropertyP] *exactly* [num] [ClassB]	LSP-OP-Exact-EN
T12. Universal relation P between the union of two classes A and B	[ClassA] *subclassOf* [PropertyP] *only* [ClassB] *or* [ClassC]	LSP-U-EN
T13. Universal relation P between the intersection of two classes A and B	[ClassA] *subclassOf* [PropertyP] *some* [ClassB] *and* [ClassC]	LSP-INTER-EN
T14. Individual I exists	[IndividualI] *type* [ClassA]	Ontology experts
T15. Property P has domain class A	[PropertyP] *domain* [ClassA]	Ontology experts
T16. Property P has range class A	[PropertyP] *range* [ClassA]	Ontology experts
	Composed tests	
T17. Multiple inheritance of class A	[ClassA] *subclassOf* [ClassB] *and* [ClassC]	LSP-MI-EN
T18. Subsumption and relation between classes	[ClassA] *subClassOf* [ClassB] *that* [PropertyP] *some* [ClassC]	LSP-DC-SC-EN
T19. Minimum cardinality between classes A and B, and existential relation P between classes B and C	[ClassA] *subclassof* [PropertyP] *min* [num] [ClassB] *and* [ClassB] *subclassof* [PropertyP] *some* [ClassC]	LSP-Min-and-OP-EN
T20. Minimum cardinality between classes A and B and universal relation P between classes B and C	[ClassA] *subclassof* [PropertyP] *min* [num] [ClassB] *and* [ClassB] *subclassof* [PropertyP] *only* [ClassC]	LSP-Min-and-OP-EN
T21. Subsumption relation between A and B, subsumption relation between A and C, and disjointness between B and C	[ClassB] *subclassOf* [ClassA] *and* [ClassC] *subclassOf* [ClassA] *that disjointWith* [ClassB]	LSP- SC-Di-EN
	ODP-based tests	
T22. Participation ODP between classes A and B	[ClassA] *subClassOf isParticipantIn—hasParticipant some* [ClassB]	LSP-PA-EN
T23. Co-participation ODP	[ClassA] *and* [ClassB] *subClassOf isParticipantIn—hasParticipant some* [ClassC]	LSP-PCP-EN
T24. PartOf ODP between classes A and B, with existential restriction	[ClassA] *subClassOf isPartof—hasPart some* [ClassB]	Ontology experts
T25. PartOf ODP between classes A and B, with universal restriction	[ClassA] *subClassOf isPartof—hasPart only* [ClassB]	Ontology experts
T26. Object-Role ODP between classes A and B, with existential restriction	[ClassA] *subClassOf isRoleOf—hasRole some* [ClassB]	LSP-OR-EN
T27. Object-Role ODP between classes A and, with universal restriction	[ClassA] *subClassOf isRoleOf—hasRole only* [ClassB]	LSP-OR-EN
	Usage tests	
T28. Two individuals of classes A and B can be related by property P	[ClassA] [PropertyP] [ClassB]	Ontology experts

- **Simple tests**. These tests are associated with a single axiom in an ontology. These axioms could be related to terms declarations as well as to restrictions.
- **Composed tests**. These tests are associated with a combination of axioms in an ontology, which could be a combination of class axioms, descriptions, or property axioms.
- **ODP-based tests**. These tests are associated with ODPs. Up to now, the ODP-based test expressions are associated with Content Ontology Design Patterns (CP) [16], which propose patterns for solving design problems for the domain classes and properties that populate an ontology and, therefore, for addressing content problems. Examples of these CPs are the Participation[10] or the PartOf[11] ones.
- **Usage tests**. These tests are related to how the ontology is used, rather than checking a particular restriction in the ontology.

[10] http://ontologydesignpatterns.org/wiki/Submissions:Participation.
[11] http://ontologydesignpatterns.org/wiki/Submissions:PartOf.

During this activity, the goal of each requirement is extracted and formalised using the set of test expressions included in the catalogue described in Table 4. To create the test, each ontology requirement should be associated with at least one test expression presented in Table 4. Each test expression is related to a particular goal to be checked by such a test, as shown in Table 4, which should be in agreement with the goal of the requirement. A complex requirement that includes several sentences could be categorised with more than one LSP and, therefore, could be associated with more than one test.

The tests are defined without any information, e.g., URIs or labels, related to the ontology in which such a test will be executed. With this separation between the test and the ontology, the reuse of tests in different ontologies is allowed. However, the terms included in the test must be present in the glossary of terms of the ontology on which the test will be executed, as shown in Fig. 1. This glossary of terms aims to map these terms that are defined in the test design to terms in the ontology. The terms must follow camel-case style, e.g., *DigitalEntity*.

As an example of test design, the requirement that states "*An IoT gateway is a digital entity*" has as goal to model a subsumption relation between two entities. Therefore, the test to be used should refer to "T2: Subsumption relation between classes A and B". Consequently, the test expression for T2 determined in the catalogue, i.e., "[ClassA] *subClassOf* [ClassB]", is completed with the information of the requirement. The requirement "*An IoT gateway is a digital entity*" is then associated with the test "Gateway *subClassOf* DigitalEntity", since Gateway and DigitalEntity are terms that describe the concepts in the test. Then, the terms Gateway and DigitalEntity must be included into the glossary of terms as keywords to be associated with terms in the ontology during the test execution activity, e.g., Gateway and DigitalEntity could be associated with the terms in the ontology named <http://www.example.org/ontology#Gateway> and <http://www.example.org/ontology#DigitalEntity>, respectively.

3.2.2. Test implementation activity

During this activity, the tests should be implemented to be executed on an ontology. To this end, each test is formalised into a precondition, a set of auxiliary term declarations, and a set of assertions to check the behaviour. This testing method proposes a test implementation for each test design included in Table 4. Therefore, the implementation of the proposed test design can be automated.

The **precondition** is a SPARQL query that checks whether the terms involved in the test are defined in the ontology. 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** (i.e., test preparation) are a set of temporary axioms added to the ontology to declare the auxiliary terms needed to carry out the assertions.

Table 5 shows the test implementation associated with the test design for checking the equivalence between classes (test expression with identifier T4 in Table 4). For the sake of readability, the axioms in these tables are represented by means of the Description Logics syntax [2].

To check the equivalence between two classes, a set of auxiliary terms are defined, i.e., the classes that complement A (i.e., $\neg A$) and B (i.e., $\neg B$). After their definition, a set of assertions that force the ontology to present unsatisfiable classes or inconsistencies are also defined. The first, associated with axiom 'E 2' in Table 5, 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 a subclass of A and $\neg A$. The second assertion, associated with axiom 'E 3', 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 a subclass of B and $\neg B$. The last assertion, associated with axiom 'E 4', generates a class A' that is defined as a subclass of classes A and B. If the ontology satisfies the equivalence requirement, this assertion causes a consistent ontology due to the fact that there is no problem if A' is a subclass of A and B.

3.2.3. Test execution activity

Taking as input the test implementation, the test execution activity consists of three steps: (1) the execution of the query that represents the preconditions, (2) the addition of the axioms that declare the auxiliary terms, and (3) the addition of the assertions. After the addition of each axiom, a reasoner is executed to report the status of the ontology, i.e., whether the ontology is consistent, inconsistent, or has unsatisfiable classes. The addition of auxiliary axioms should always lead to a consistent ontology. However, in the case of assertions, the agreement between the reasoner status after the addition of all axioms and the status indicated in the test implementation determines whether the ontology satisfies the desired behaviour and, consequently, the requirement. The steps carried out during the execution activity are summarised in Algorithm 1.

During this activity, the test implementation should be first completed with the information related to the ontology to be executed. To that end, a glossary of terms must be generated manually or automatically. As mentioned before, the glossary of terms maps each term in the test to a term in the ontology to be analysed. Therefore, the terms that are defined in the ontology, e.g., *Gateway*, are collected and associated with a URI in the ontology, e.g., <http://www.example.org/ontology#Gateway>. Then, using these associations, the terms in the test implementation are translated into terms in the ontology. Consequently, these test implementations can then be executed on the ontology. This requires that the terms in the test expressions be included in the glossary. However, it is possible that a term in the test expression is not included in the glossary of terms of the ontology. In this case, the ontology does not include the terms asked by the test and, therefore, the test is not passed.

There are four possible results of the execution step for each test and each ontology:

Table 5
T4. Equivalence between two classes A and B.

Goal:	T4. Equivalence between two classes A and B		
Test expression:	[ClassA] equivalentTo [ClassB]		
Type:	Simple test	**Related to:**	Classes
Test precondition		**Test preparation**	
Class A and Class B exist		(E 1.1) Declaration of ¬A (E 1.2) Declaration of ¬B	
Assertions to test the ontology behaviour			
Axiom		**Result**	
(E 2) AssertionA' $\sqsubseteq \neg A \sqcap B$		Unsatisfiable class	
(E 3) Assertion A' $\sqsubseteq A \sqcap \neg B$		Unsatisfiable class	
(E 4) Assertion A' $\sqsubseteq A \sqcap B$		Consistent ontology	

1. **Passed**: if the ontology passes, the preconditions and the results of the assertions are the expected ones.
2. **Undefined term**: if the ontology does not pass the preconditions, i.e., some of the terms in the test expression are not defined in the ontology.
3. **Absent**: if the ontology passes the preconditions and the results of the assertion are not the expected ones but there are no conflicts in the ontology, i.e., there is a missing relation between terms in the ontology.
4. **Conflict**: if the ontology passes the preconditions and the results of the assertion are not the expected ones, and the addition of the axioms related to the test expression leads to a conflict in the ontology.

Algorithm 1. Test execution

Input: Ontology and test implementation
Output: Test result

```
loadOntology(ontology);
if (!preconditionsSatisfied(ontology, test.preconditions)) then
    return UNDEFINED_TERM;
end if
add(ontology, test.auxiliaryTerms);
if (ontologyStatus(ontology) ≠ consistent)) then
    return CONFLICT;
end if
for all assertion in test.assertions do
    loadOntology(ontology);
    add(ontology, test.auxiliaryTerms);
    add(ontology, assertion.axioms);
    status = ontologyStatus(ontology);
    if (status ≠ assertion.result) then
        if (status = inconsistent OR unstatisfiable) then
            return CONFLICT;
        else
            return ABSENCE;
        end if
    end if
end for
return PASSED;
```

3.3. Traceability between tests, requirements, and ontologies

An ontology for modelling tests could provide not only a guide on how to create tests for ontology verification but also a procedure for creating reusable tests and for allowing traceability between ontologies, requirements, tests, and test results. In addition, the test suite written following the ontology with all the information related to the requirements and tests can be considered as a formalised documentation. Having all this information stored in the RDF format could also allow having a

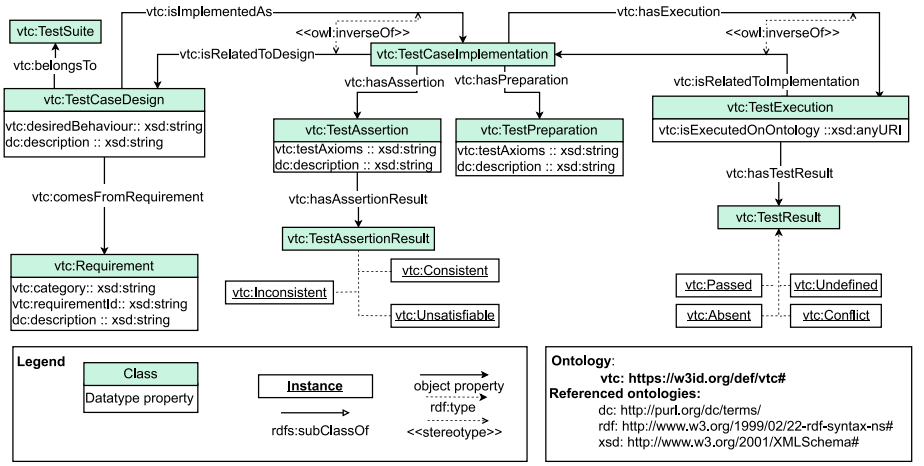


Fig. 2. Overview of the Test Case Verification ontology.

knowledge graph on which users could execute queries to obtain information about which requirements are satisfied for each ontology. For this reason, the Test Case Verification ontology,[12] which is shown in Fig. 2, has been developed in the context of the ontology verification testing method.

This ontology includes the vtc:TestCaseDesign class, which defines the design of a test. This class has several datatype properties, including description, desired behaviour (i.e., the test), and related requirements, in order to enable traceability between the test and the requirements from which it is extracted. A vtc:TestCaseDesign belongs to a particular vtc:TestSuite, which is extracted from a given source (e.g., a manual, a standard, or a list of requirements of an ontology). The vtc:TestCaseDesign can also be associated with the vtc:Requirement from which it is extracted, which has a category, an identifier, and a description. Besides, the ontology also includes the vtc:TestCaseImplementation class, which defines how each test design should be implemented. Each vtc:TestCaseImplementation is related to a vtc:TestCaseDesign and includes a vtc:TestPreparation, which represents the temporary axioms added to declare the auxiliary terms, and a vtc:TestAssertion, which represents the axioms to be added to represent each ontology scenario. Each vtc:TestAssertion has a vtc:AssertionResult, which is represented by means of three individuals, namely (1) vtc:Unsatisfiable, (2) vtc:Inconsistent and (3) vtc:Consistent. Finally, the vtc:TestExecution class defines the execution of a test implementation on a particular ontology. Therefore, it is related to a vtc:TestCaseImplementation and an ontology. A vtc:TestCaseResult has also associated a vtc:TestResult, which represents the result of the test execution. The result is represented by means of four individuals, namely (1) vtc:Passed, (2) vtc:Undefined, (3) vtc:Absent and (4) vtc:Conflict.

Listing 1 presents a short illustrative example of a test design, generated from a requirement that states "*An IoT gateway is a digital entity*", which is classified into one requirement type: "T2. Subsumption relation between classes A and B". Because this test does not have URIs related to the ontology in which the tests are going to be executed, it can be reused in other ontologies.

```
@prefix test: <http://www.example.org/testsuite#> .
@prefix dc: <http://purl.org/dc/elements/1.1/title/> .
@prefix vtc: <https://w3id.org/def/vtc#> .


test:platform2 a vtc:Requirement;
        vtc:category "Gateway";
        dc:description "An IoT gateway is a digital entity";
        vtc:requirementId "onto001".

test:testDesignPlatform2 a vtc:TestCaseDesign;
        vtc:comesFromRequirement test:platform2;
        vtc:desiredBehaviour "Gateway subClassOf DigitalEntity".
```

This traceability between the requirements, tests and ontology allows users to interact with the verification results by, for example, querying the data to obtain:

- Which ontologies satisfy a particular requirement.
- Which requirements are defined for a particular ontology.
- Which tests are related to a particular concept.

[12] The Verification Test Case ontology is available online: https://w3id.org/def/vtc.

As an example of use, Themis publishes the set of test designs available on its website[13] using the Helio tool,[14] which also provides a SPARQL endpoint[15] to execute queries on the dataset. As an example of SPARQL query in this dataset, Listing 2 shows how to search for those tests that are defined for the WoT ontology.[16]

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
CONSTRUCT{
    ?test ?source ?ontology
} WHERE{
    ?testsuite ?type  <http://w3id.org/def/vtc#TestSuite> .
    ?testsuite <http://purl.org/dc/terms/source> <http://iot.linkeddata.es/def/wot> .
    ?test <http://w3id.org/def/vtc#belongsTo> ?testsuite .
    ?test ?source ?ontology .

}
```

3.4. Technological support

Themis[17] [15] is an on-line testing tool intended to help ontology engineers and practitioners during the ontology verification activity. Themis supports and automates the test implementation and execution steps defined in Section 3.2, and implements the 28 test expressions described in Section 3.2. It provides both a web-based human interface and a REST API to be used by applications.

Themis is implemented in Java and uses well-known open-source frameworks and libraries. The code of the tool is available in Github[18] under Apache License 2.0.[19]

4. Evaluation

This section describes the empirical evaluation performed to validate the research questions and hypotheses presented in Section 1, which are focused on the comparison between tools that use testing languages and tools that do not with regard to errors and response time during the verification activity.

With the aim of determining whether the use of a testing language for defining tests facilitates the ontology testing method in terms of errors and time, Themis, the tool presented in Section 3.4 that supports the verification testing method described in Section 3, was compared through a user evaluation with other tools of the state of the art for ontology verification. Some of these tools also use a testing language for the definition of tests.

Different participants with different ontology development expertise participated in such evaluation, providing feedback regarding the verification tools and their usability.

4.1. Experimental design

The goal of this analysis was to compare ontology verification tools that use testing languages with tools that do not. To this end, each participant in this experiment had to verify whether a given ontology satisfied a set of ontology requirements with one of these tools. Depending on the tool, participants had to execute tests or to browse the terms and axioms in the ontology to check whether the requirements were satisfied.

For this evaluation, a set of 30 ontology requirements written in natural language was defined and, based on them, an ontology was developed and published.[20] These requirements included diverse restrictions and had different complexities, such as simple requirements that require a hierarchy and complex requirements associated with cardinality or with several relations between terms. Moreover, only some of the requirements were satisfied by the ontology. The participants had to identify which of them were satisfied by the ontology and which were not. The list of requirements used in this experiment is shown in Table 6.

Based on the results of the participants, the verification tools can be compared according to two aspects, namely, the errors made by the participants and the time spent in the verification process.

To check the errors made by the participants, the results provided, i.e., the set of satisfied or unsatisfied requirements, were analysed. To identify for each tool:

- *Incorrect answers*, i.e., to affirm that a requirement is satisfied by an ontology when it is not or vice versa.

[13] http://themis.linkeddata.es/catalogue.html.
[14] https://oeg-upm.github.io/helio/.
[15] https://helio.vicinity.iot.linkeddata.es/sparql.
[16] http://iot.linkeddata.es/def/wot.
[17] http://themis.linkeddata.es.
[18] https://github.com/oeg-upm/Themis.
[19] https://www.apache.org/licenses/LICENSE-2.0.
[20] https://w3id.org/def/themisEval#.

Table 6
List of requirements.

Identifier	Type	Requirement	Satisfied
R.1	Hierarchy	A smoke sensor is a type of sensor	Yes
R.2	Disjointness	Open commands differ from close commands	No
R.3	Terms relation	A switch consists of only an actuator	No
R.4	Hierarchy	A meter is a type of device and physical thing	No
R.5	Disjointness	A sensor cannot be an actuator	No
R.6	Terms relation	An agent can only interact using an application	No
R.7	Cardinality	A device has one or more functions	Yes
R.8	Instance	Drying is an example of task	Yes
R.9	Cardinality	Each task is accomplished by at least 2 devices	No
R.10	Cardinality	A service is offered by exactly 1 device	No
R.11	Terms relation and union	A device can only have a typical consumption of energy or power	Yes
R.12	Terms relation and hierarchy	An actuator, which is a type of device, has an actuating function	Yes
R.13	Terms relation	An illuminance unit observes some light	No
R.14	Hierarchy and disjointness	Among the different types of commands there are: open command and stop command; which are also disjoint	No
R.15	Cardinality and terms relation	A device has one or more functions and these functions accomplish some tasks	No
R.16	Cardinality and terms relation	A sensor measures at least one property and these properties relate to measurement	No
R.17	Hierarchy	Devices are divided into two different types: actuator and sensor	Yes
R.18	Terms relation	An agent participates in an organisation	Yes
R.19	Terms relation	An agent is a participant in an event	No
R.20	Instance	The accumulated energy produced is an example of measurement	No
R.21	Hierarchy	A sensor is a device	Yes
R.22	Hierarchy	A sensor is a physical thing	No
R.23	Cardinality	A sensor has to accomplish at least 1 task	Yes
R.24	Cardinality	A sensor must make at least 1 measurement	No
R.25	Terms relation	A light switch can consist only of several devices	No
R.26	Terms relation	A washing machine consists of some sensor	No
R.27	Disjointness	An agent cannot be a device	No
R.28	Disjointness	A temperature sensor cannot be an actuator	No
R.29	Cardinality	An indicator relates to at least 2 properties	No
R.30	Cardinality	Power is controlled by exactly 1 device	No

- *Correct answers*, i.e., to affirm that a requirement is not satisfied by an ontology and it is not or that a requirement is satisfied by an ontology and it is.
- *Unsolvable requirements*, which are requirements that could not be verified by participants. In this analysis, participants were asked to leave a requirement as 'unsolvable' if they had to spend more than 5 min to check if it was satisfied or not. In addition, unsolvable requirements refer to those requirements that were not understood by the participants.

To check the time spent by the participants for each tool during the experiment, the time spent by each participant in verifying each requirement was calculated.

With the aim of comparing the tools, a web application with an online questionnaire was developed to be completed by the participants in the experiment. This questionnaire collected data from participants and their results regarding the verification process.

First, the questionnaire asked participants to add their demographic data, including expertise in the OWL language and software development. Furthermore, participants were asked about their previous experience with the tools analysed in the experiment.

Subsequently, the questionnaire asked the participants about the verification process for each requirement in the experiment. The following information had to be added by each participant for each requirement:

- The test or technique that was used to verify the requirement (e.g., the use of a reasoner or an editor browser).
- Based on the results of the test or the technique used, participants had to indicate for each requirement: (1) if the requirement is satisfied; (2) if there are terms in the requirement that are not included in the ontology; (3) if there is any absent restriction; (4) if there is a conflict between the requirement and the analysed ontology; or (5) if the participant does not know how to verify the requirement.
- Feedback or comments that the participants wanted to report related to the tool, the requirement, or the tests associated with it.

In addition, the questionnaire automatically collects the time spent by each participant in identifying whether each requirement is satisfied by the ontology or not.

Once the participants completed the questionnaire, a USE questionnaire [25] was sent to them with questions related to usefulness, satisfaction, and usability of the tools. The questionnaire included 11 qualitative questions divided into three categories, namely, usefulness, satisfaction, and usability. Within the questionnaire, the questions are rated on a Likert scale from 1 (strongly disagree) to 5 (strongly agree). The questionnaire also included questions for listing the most positive and negative aspects of the tools and allowed participants to send additional comments regarding the usability of the tools.

4.2. Results and discussion

To perform this experiment, participants used the tools Themis (described in Section 3.4), TDDOnto2 (described in Section 2) and Protégé.[21] Both Themis and TDDOnto2 are tools that use testing languages to define tests to verify an ontology. As mentioned in the previous sections, Themis uses the testing language presented in Section 3.2, which is based on the OWL Manchester Syntax, and provides guides to help users in the definition of tests. Regarding TDDOnto2, its tests are based on axioms written in the OWL Manchester syntax. TDDOnto2 relies on users' knowledge and, therefore, no guides are provided to help users, although it includes an autocompletion service. Protégé is an ontology editor that allows visualising and browsing the hierarchy of classes and their properties in an ontology.

In this evaluation, 30 participants were involved (10 participants per tool), among which there were experts and non-experts in the OWL language (participants that were familiar and not familiar with OWL). Participants in the experiment are required to have a minimum of knowledge in Semantic Web technologies. The set of participants also included undergraduate and master students. Tables 7 and 8 summarise the expertise of the participants in ontology and software development, as well as their expertise in the tools.

As shown in these tables, most of the participants are familiar with the OWL language, but they do not have a background in the OWL Manchester Syntax. In addition, there are several participants who are also familiar with software development, which can help to understand the testing process. Finally, the majority of the participants had not used any of the testing tools until this experiment.

4.2.1. Time analysis

The experiment was planned to last around one hour and a half. It should be noted that not all participants answered the 30 requirements due to time problems, since they spent more than the planned time.

Figs. 3–5 show the obtained results with respect to the time spent per requirement in the experiment, indicating the average seconds spent per requirement in each of the tools. It should be noted that in Fig. 3 Themis does not have values from requirements R.14 to R.30. This is because the only participant not familiar with the OWL language that used Themis could not complete the questionnaire due to time problems.

From these figures, it can be observed that, regardless of the expertise of the participants, Protégé has a stable time spent per requirement during the verification process, while both Themis and TDDOnto2 have a learning curve. As an example, in Fig. 5 it can be observed that R.1 and R.13 for TDDOnto2 take more than twice the time spent in Protégé for participants experts in the OWL language. However, after R.19 the time spent in the three tools is similar, although the complexity of these requirements is not different from the previous ones. This evolution could indicate the existence of the learning curve in both TDDOnto and Themis. Furthermore, it can be observed that even if the time is higher during the first requirements, it decreases over time for all participants.

4.2.2. Correctness analysis

During this experiment, the answers (incorrect and correct) given by the users were also collected. Checking the summarised results presented in Table 9, it can be concluded that the percentage of correct results is significantly higher in Themis and TDDOnto2 for users that are familiar or experts in OWL. For this range of users, i.e., familiar and experts, Themis and TDDOnto2 also present a significantly lower number of incorrect results.

For those users that were not familiar with OWL, there is not a high variability in the results of the tools (even if they are different). It should also be noted that Protégé had more participants that are not familiar with the OWL language than the other tools.

Tables 10–12[22] detail the results collected from the analysis, grouped by the type of participant and by the tool. These figures include the percentage of correct and incorrect answers for each requirement and the number of unsolved requirements, i.e., those requirements that the participants did not know how to verify.

From these results, it can be concluded that, although Protégé is the tool with the most stable time spent per requirement, it is also the tool with the worst results regarding the errors made during the verification process. In contrast, Themis and TDDOnto2 have a learning curve, but yield better results in terms of correctness for those users that are familiar and experts in OWL.

[21] https://protege.stanford.edu/.

[22] For the sake of readability, the numbers of all tables in this section are rounded. In addition, C refers to correct results, I to incorrect results and U to unresolvable results. The sum of the percentages of C, I and U needs to be 100%. A 0% value refers to the fact that there are 0 requirements with correct, incorrect, or unsolvable results, depending on the column.

Table 7
OWL expertise of participants in the experiment.

Tool	OWL expertise			OWL Manchester Syntax expertise		
	Not familiar	Familiar	Expert	Not familiar	Familiar	Expert
Themis	1	7	2	4	6	0
TDDOnto2	1	6	3	8	1	1
Protégé	6	3	1	8	1	1

Table 8
Development expertise of participants in the experiment.

Tool	Software development expertise			Tool expertise		
	Not familiar	Familiar	Expert	Not familiar	Familiar	Expert
Themis	1	7	2	9	1	0
TDDOnto2	4	3	3	8	2	0
Protégé	4	4	2	5	3	2

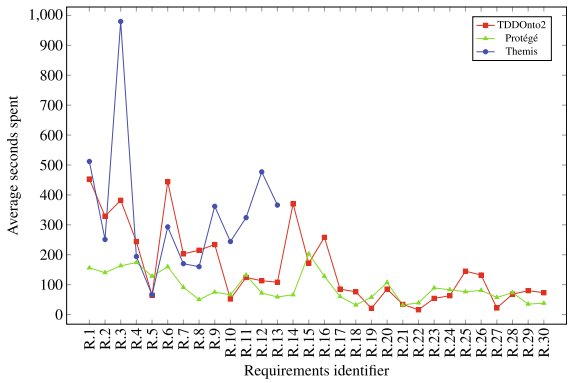


Fig. 3. Comparison of the time spent per requirement for participants not familiar with the OWL language.

Apart from this, it should be noted that in the three tools there were some requirements that the participants could not test, although the number of these unsolved requirements is low. The requirements that had a higher percentage of unresolved results were R.12, R.13, R.16, and R.25 in the case of Themis, R.13, R.15, R.16 and R.17 in the case of TDDOnto2 and R.20, R.25 in the case of Protégé. Requirements R.12, R.13, and R.25 are associated with relations between terms in the ontology, R.15 and R.16 also include a cardinality restriction, R.17 is related to a hierarchy between classes, and R.20 is associated with the definition of an instance of a class. This motivates the analysis of the correct results in terms of the type of requirement.

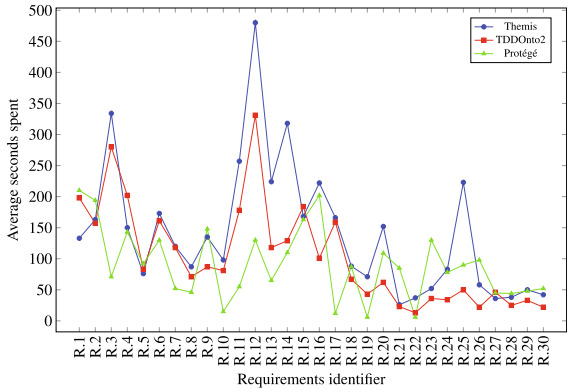


Fig. 4. Comparison of the time spent per requirement for participants familiar with the OWL language.

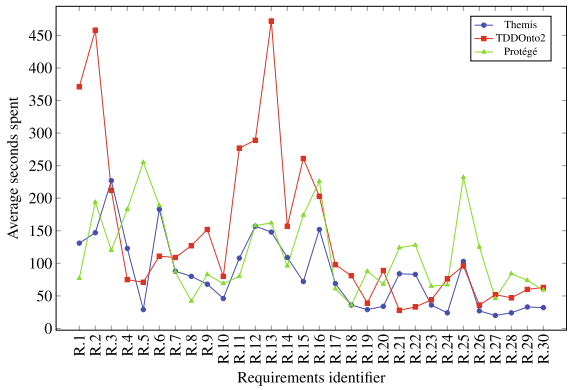


Fig. 5. Comparison of the time spent per requirement for participants experts with the OWL language.

Most of the requirements with a higher percentage of unresolved results are related to term relations since the participants could not identify which restrictions, e.g., existential or universal ones, were associated with such requirements.

Tables 13–15 show the results obtained from the three tools grouped by type of requirement. From these tables, it can be observed that, in general, those requirements that include several restrictions, e.g., hierarchy and disjoint classes, are the most difficult requirements to verify, since they have a higher percentage of incorrect results.

Regarding Themis, for the participants who were not familiar with OWL, the requirements that had the highest percentage of correct answers were those related to instances or related both to a relation and a hierarchy between terms, which had 100% of correct results. For the participants who were familiar with OWL, the requirements related to disjoint classes had the highest percentage of correct answers (i.e., 100% of correct results). Finally, for the participants that were experts, the requirements that had the highest percentage of correct answers were those related to disjoint classes and to relations between terms, and those requirements that had more than one restriction such as a relation and a hierarchy or union between classes. These three types of requirements also had 100% of correct results.

Concerning incorrect results, in the case of Themis, the requirements with the lowest percentage of correct answers for the participants that were not familiar with OWL were those related to hierarchies and to relations and unions between terms, which both had 100% of incorrect results. Regarding participants who were familiar with the OWL language, those requirements related to relations and hierarchies had the lowest percentage of correct answers, although it is higher than 50%. Finally, regarding OWL experts, the requirements with the lowest percentage of correct answers were those related to hierarchies and disjoint classes, which had 50% of incorrect results.

In the case of TDDOnto2, the highest percentage of correct answers for the participants that were not familiar with OWL were related to instances, cardinality and hierarchy and disjointness. All these requirements had 100% of correct results. Concerning participants who were familiar with the OWL language, the requirements related to hierarchy and the requirements that define both hierarchies and disjoint classes were the ones with the highest percentage. These requirements had 86% and 83% of correct results, respectively. Finally, concerning experts, the requirements related to instances, as well as those related to both relations and hierarchies, had 100% of correct answers.

Regarding incorrect results, for TDDOnto2, the requirements with the lowest percentage of correct results for all the participants were those related to several restrictions. For participants who were not familiar with OWL, the requirements related to relations and cardinality between terms, to relations between terms and hierarchies, and to relations and unions between terms, had 0% of correct answers. Regarding participants that were familiar with OWL, the requirements related to relations and unions between terms had the lowest percentage of correct answers. Finally, for those participants experts in

Table 9
Summary of the error results.

	Participants								
	Not Familiar			Familiar			Expert		
	C	I	U	C	I	U	C	I	U
				Themis					
Average results	46%	54%	8%	79%	18%	2%	83%	17%	0%
				TDDOnto2					
Average results	68%	10%	23%	72%	24%	6%	81%	16%	0%
				Protégé					
Average results	60%	40%	2%	43%	57%	0%	50%	50%	0%

Table 10
Percentage of correctness results for Themis.

Requirement	Participants								
	Not familiar			Familiar			Expert		
	C	I	U	C	I	U	C	I	U
R.1	0%	100%	0%	50%	50%	0%	50%	50%	0%
R.2	0%	100%	0%	100%	0%	0%	100%	0%	0%
R.3	0%	100%	0%	100%	0%	0%	100%	0%	0%
R.4	0%	100%	0%	100%	0%	0%	100%	0%	0%
R.5	100%	0%	0%	100%	0%	0%	100%	0%	0%
R.6	0%	100%	0%	71%	29%	0%	100%	0%	0%
R.7	100%	0%	0%	71%	29%	0%	100%	0%	0%
R.8	100%	0%	100%	43%	58%	0%	100%	0%	0%
R.9	0%	100%	0%	71%	29%	0%	50%	50%	0%
R.10	100%	0%	0%	86%	14%	0%	100%	0%	0%
R.11	0%	100%	0%	100%	0%	0%	100%	0%	0%
R.12	100%	0%	0%	42%	42%	14%	50%	50%	0%
R.13	100%	0%	0%	57%	29%	14%	50%	50%	0%
R.14	–	–	–	86%	14%	0%	50%	50%	0%
R.15	–	–	–	100%	0%	0%	50%	50%	0%
R.16	–	–	–	71%	0%	29%	100%	0%	0%
R.17	–	–	–	0%	100%	0%	50%	50%	0%
R.18	–	–	–	100%	0%	0%	100%	0%	0%
R.19	–	–	–	100%	0%	0%	100%	0%	0%
R.20	–	–	–	71%	29%	0%	50%	50%	0%
R.21	–	–	–	86%	14%	0%	0%	100%	0%
R.22	–	–	–	100%	0%	0%	100%	0%	0%
R.23	–	–	–	86%	14%	0%	100%	0%	0%
R.24	–	–	–	86%	14%	0%	100%	0%	0%
R.25	–	–	–	14%	71%	14%	100%	0%	0%
R.26	–	–	–	100%	0%	0%	100%	0%	0%
R.27	–	–	–	100%	0%	0%	100%	0%	0%
R.28	–	–	–	100%	0%	0%	100%	0%	0%
R.29	–	–	–	100%	0%	0%	100%	0%	0%
R.30	–	–	–	86%	14%	0%	100%	0%	0%
Average	**46%**	**54%**	**8%**	**79%**	**18%**	**2%**	**83%**	**17%**	**0%**

OWL, the requirements related to relations and cardinality, to hierarchy and disjointness, and to relations and unions between terms had 67% of correct answers, which was the lowest percentage of correct answers for this type of participants.

For Protégé, the type of requirement with the highest percentage of correct answers for all the participants was the one related to relations and unions between terms, which was not a trivial requirement for participants using the other tools. This could happen due to the visualisation of the restrictions provided by Protégé.

The requirements related to a hierarchy in Protégé had a low percentage of correct answers compared to the other tools. For participants that are familiar with OWL only 33% of this type of requirement had correct answers, while for experts only 40%. This situation might arise because Protége only shows the direct superclass and, consequently, users cannot see the entire hierarchy unless they navigate through the classes. This might also happen with those requirements related to disjoint classes and to relations between terms.

Table 16 summarises the results obtained per type of requirement grouped by tool, as well as the total percentage of results for all tools. With the information depicted in Table 16, it is confirmed that the analysed tools had more than 50% of correct results on average for all types of requirements. Moreover, it is also confirmed that, on average, the requirements more complex to be verified, i.e., with the worst percentage of incorrect results, were those related to terms relations and hierarchies. It should also be noted that the number of unsolvable requirements for the three tools is very low. As shown in Table 16, for almost all the types, the number of unsolvable requirements is 0.

It must also be remarked that the variability in the results for the different tools is high. This makes sense because the testing approaches provided by each tool facilitate dealing with certain types of test above others. For example, the percentage of incorrect results regarding disjointness is quite low for Themis, since the tool only requires writing a simple test expression; on the other hand, checking disjointness with Protégé requires browsing the disjointness axioms in the ontology and even running a reasoner when those axioms are not explicit.

It was also analysed using a correlation coefficient [21] whether the average time spent per requirement could influence the correctness of the results. Taking into account the information presented in Tables 10–12 and in Figs. 3–5, it was concluded that there is no significant correlation between the time and the errors in this experiment.

Table 11
Percentage of correctness results for TDDOnto2.

Requirement	Participants								
	Not familiar			Familiar			Expert		
	C	I	U	C	I	U	C	I	U
R.1	50%	50%	0%	100%	0%	0%	100%	0%	0%
R.2	100%	0%	0%	83%	17%	0%	67%	33%	0%
R.3	100%	0%	0%	67%	33%	0%	100%	0%	0%
R.4	100%	0%	0%	100%	0%	0%	100%	0%	0%
R.5	0%	50%	0%	0%	100%	0%	33%	67%	0%
R.6	0%	0%	100%	67%	33%	0%	67%	33%	0%
R.7	100%	0%	0%	100	0%	0%	100%	0%	0%
R.8	100%	0%	100%	67%	33%	0%	100%	0%	0%
R.9	100%	0%	0%	50%	50%	0%	67%	33%	0%
R.10	100%	0%	0%	83%	17%	0%	100%	0%	0%
R.11	0%	100%	0%	17%	83%	0%	67%	33%	0%
R.12	0%	0%	100%	67%	33%	0%	100%	0%	0%
R.13	100%	0%	0%	67%	17%	17%	33%	67%	0%
R.14	100%	0%	0%	83%	17%	0%	67%	33%	0%
R.15	0%	100%	0%	33%	33%	33%	33%	67%	0%
R.16	0%	0%	100%	83%	0%	17%	100%	0%	0%
R.17	0%	0%	100%	33%	50%	17%	67%	33%	0%
R.18	100%	0%	0%	50%	50%	0%	67%	33%	0%
R.19	100%	0%	0%	100%	0%	0%	100%	0%	0%
R.20	100%	0%	0%	80%	20%	0%	100%	0%	0%
R.21	100%	0%	0%	100%	0%	0%	100%	0%	0%
R.22	100%	0%	0%	100%	0%	0%	100%	0%	0%
R.23	100%	0%	0%	80%	20%	0%	100%	0%	0%
R.24	100%	0%	0%	100%	0%	0%	100%	0%	0%
R.25	0%	0%	100%	40%	60%	0%	67%	33%	0%
R.26	0%	0%	100%	100%	0%	0%	100%	0%	0%
R.27	100%	0%	0%	100%	0%	0%	100%	0%	0%
R.28	100%	0%	0%	80%	20%	0%	100%	0%	0%
R.29	100%	0%	0%	80%	20%	0%	100%	0%	0%
R.30	100%	0%	0%	80%	20%	0%	100%	0%	0%
Average	**68%**	**10%**	**23%**	**72%**	**24%**	**6%**	**81%**	**16%**	**0%**

4.2.3. Usefulness, satisfaction, and ease of use analysis

After the empirical analysis, the information collected through the USE questionnaire was analysed. To this end, 30 questionnaires (10 questionnaires for each tool) were collected from the participants. In Figs. 6–8 an overview of the results is provided.

Figs. 6–8 show that with respect to usefulness, satisfaction, and ease of use, Themis had the best results, followed by Protégé and TDDOnto2. However, although Protégé had good results on the questionnaire, especially as an easy to use and intuitive tool, it leads to worse results in the verification process. From the analysis and feedback from participants, it was also found that for those participants without any knowledge related to the OWL language and ontologies, learning a syntax to generate tests was too complex. In that scenario, Protégé is the most useful and usable tool. However, it was also found that these participants are able to search for classes and properties in an ontology, but they cannot verify other types of requirements, since it is difficult for them to understand the different types of restrictions that can be defined in an ontology. Furthermore, instead of having complex test expressions or concatenated axioms in the case of TDDOnto2, it was easier and more intuitive for the participants to use simple tests, although they had to split the test into several subtests. However, in general, all participants found these tools useful to verify ontologies.

4.3. Conclusions of the experiment

Based on the information gathered from the experiment, the following conclusions regarding the hypotheses were obtained:

- **H1. Using a testing language to define tests based on functional requirements facilitates the ontology testing process regarding the reduction of time in users who are familiar and experienced in OWL.**
  Considering time, although it is true that the learning curve is higher in Themis and TDDOnto2, once the user gets familiar with the syntax, the time spent per requirement is similar in all the analysed tools. However, Protégé is the tool with a more stable average time. For the participants that were not familiar with OWL, Protégé is the tool with a shorter average

Table 12
Percentage of correctness results for Protégé.

Requirement	Participants								
	Not familiar			Familiar			Expert		
	C	I	U	C	I	U	C	I	U
R.1	100%	0%	0%	100%	0%	0%	100%	0%	0%
R.2	83%	17%	0%	100%	0%	0%	100%	0%	0%
R.3	17%	83%	0%	0%	100%	0%	0%	100%	0%
R.4	33%	67%	0%	0%	100%	0%	0%	100%	0%
R.5	17%	83%	0%	0%	100%	0%	0%	100%	0%
R.6	67%	33%	0%	33%	67%	0%	0%	100%	0%
R.7	83%	17%	0%	100%	0%	0%	100%	0%	0%
R.8	83%	17%	0%	33%	67%	0%	100%	0%	0%
R.9	50%	50%	0%	100%	0%	0%	100%	0%	0%
R.10	17%	83%	0%	0%	100%	0%	100%	0%	0%
R.11	83%	17%	0%	100%	0%	0%	100%	0%	0%
R.12	67%	33%	0%	33%	67%	0%	0%	100%	0%
R.13	83%	17%	0%	67%	33%	0%	0%	100%	0%
R.14	67%	33%	0%	67%	33%	0%	100%	0%	0%
R.15	67%	33%	0%	0%	100%	0%	0%	100%	0%
R.16	83%	17%	0%	67%	33%	0%	100%	0%	0%
R.17	0%	100%	0%	0%	100%	0%	0%	100%	0%
R.18	100%	0%	0%	100%	0%	0%	0%	100%	0%
R.19	83%	67%	0%	33%	67%	0%	0%	100%	0%
R.20	50%	33%	17%	67%	33%	0%	0%	100%	0%
R.21	100%	0%	0%	67%	33%	0%	0%	100%	0%
R.22	33%	67%	0%	0%	100%	0%	100%	0%	0%
R.23	100%	0%	0%	33%	67%	0%	100%	0%	0%
R.24	100%	0%	0%	0%	100%	0%	100%	0%	0%
R.25	33%	33%	33%	0%	100%	0%	100%	0%	0%
R.26	33%	67%	0%	67%	33%	0%	0%	100%	0%
R.27	33%	67%	0%	0%	100%	0%	0%	100%	0%
R.28	17%	83%	0%	0%	100%	0%	100%	0%	0%
R.29	67%	33%	0%	100%	0%	0%	0%	100%	0%
R.30	50%	50%	0%	33%	67%	0%	100%	0%	0%
Average	**60%**	**40%**	**2%**	**43%**	**57%**	**0%**	**50%**	**50%**	**0%**

time per requirement. For participants familiar with OWL, TDDOnto2 is the tool with a shorter average time spent per requirement and for participants experts in the OWL language, Themis is the one with a shorter average time spent. Therefore, this hypothesis is rejected.

- **H2. Using a testing language to define tests based on functional requirements facilitates the ontology testing process regarding the reduction of errors in users that are familiar and experienced in OWL.**
  Themis and TDDOnto2, both based on a testing language, had a significantly higher percentage of correct results in comparison to Protégé for users that are familiar and expert in the OWL language. Moreover, although both tools have a learning curve in terms of the time spent per requirement, they had a lower number of incorrect results for these types of users. Therefore, this hypothesis is true.
- **H3. Using a testing language to define tests based on functional requirements increases the usability of the tools during the verification activity.**
  Both Themis and Protégé had very similar results in the USE questionnaire. Regarding usefulness and satisfaction, Themis

Table 13
Percentage of results for Themis grouped by type of requirement.

Requirement	Participants								
	Not familiar			Familiar			Expert		
C	I	U	C	I	U	C	I	U	
Hierachy	0%	100%	0%	77%	23%	0%	62%	37%	0%
Disjointness	50%	50%	0%	100%	0%	0%	100%	0%	0%
Terms relation	33%	67%	0%	81%	17%	2%	100%	0%	0%
Instance	100%	0%	0%	62%	33%	5%	80%	20%	0%
Cardinality	67%	33%	0%	76%	16%	86%	90%	10%	0%
Terms relation and cardinality	–	–	–	86%	0%	14%	75%	25%	0%
Hierarchy and disjointness	–	–	–	86%	14%	0%	50%	50%	0%
Terms relation and hierarchy	100%	0%	0%	57%	14%	0%	100%	0%	0%
Terms relation and union	0%	100%	0%	71%	0%	26%	100%	0%	0%

Table 14
Percentage of results for TDDOnto2 grouped by type of requirement.

Requirement	Participants								
	Not familiar			Familiar			Expert		
C	I	U	C	I	U	C	I	U	
Hierachy	80%	0%	20%	86%	11%	4%	93%	7%	0%
Disjointness	75%	25%	0%	64%	36%	0%	75%	25%	0%
Terms relation	57%	0%	43%	72%	25%	2%	76%	24%	0%
Instance	100%	0%	0%	73%	27%	0%	100%	0%	0%
Cardinality	100%	0%	0%	82%	18%	0%	95%	5%	0%
Terms relation and cardinality	0%	0%	100%	58%	0%	42%	67%	33%	0%
Hierarchy and disjointness	100%	0%	0%	83%	17%	0%	67%	33%	0%
Terms relation and hierarchy	0%	0%	100%	60%	40%	0%	100%	0%	0%
Terms relation and union	0%	0%	100%	17%	0%	83%	67%	33%	0%

Table 15
Percentage of results for Protégé grouped by type of requirement.

Requirement	Participants								
	Not familiar			Familiar			Expert		
C	I	U	C	I	U	C	I	U	
Hierachy	53%	47%	0%	33%	67%	0%	40%	60%	0%
Disjointness	75%	25%	0%	25%	75%	0%	50%	50%	0%
Terms relation	64%	36%	0%	43%	57%	0%	14%	86%	0%
Instance	67%	25%	8%	50%	50%	0%	50%	50%	0%
Cardinality	67%	33%	0%	52%	48%	0%	86%	14%	0%
Terms relation and cardinality	75%	25%	0%	33%	67%	0%	50%	50%	0%
Hierarchy and disjointness	67%	33%	0%	67%	33%	0%	100%	0%	0%
Terms relation and hierarchy	60%	40%	0%	33%	67%	0%	0%	100%	0%
Terms relation and union	83%	17%	0%	100%	0%	0%	100%	0%	0%

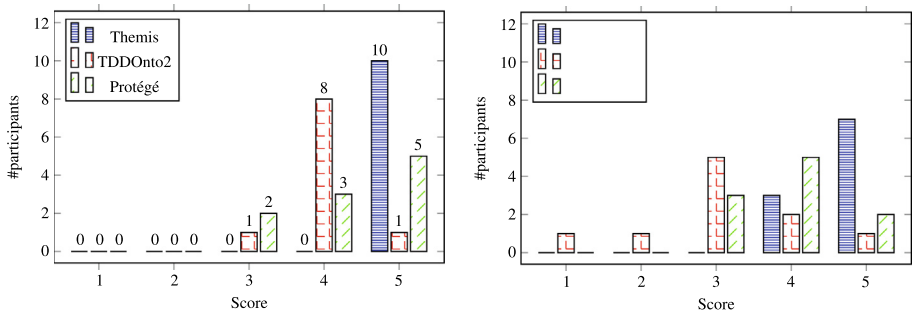
was the tool with the best results, while regarding ease of use (e.g., user friendliness or effort), Protégé had better ones. Therefore, usability does not depend only on the fact of whether the tools use testing languages or not, but also on how the user can interact with them. On the basis of these results, this hypothesis is rejected.

5. Conclusions and future work

This paper presents an ontology verification testing method that considers the participation and feedback of domain experts and users during the verification process. To define the tests, a testing language based on lexico-syntactic patterns is proposed as part of this testing method, which was defined after an analysis of how requirements are specified. Besides, following the verification testing method, the results of the testing process are proposed to be stored in a machine-readable format, which also provides traceability between the requirements, the tests, and the ontology implementation.

Table 16
Summary of the results per tool.

Requirement	Results											
	C				I				U			
	Themis	TDDOnto2	Protégé	Total	Themis	TDDOnto2	Protégé	Total	Themis	TDDOnto2	Protégé	Total
Hierarchy	73%	88%	46%	66%	27%	8%	54%	32%	0%	4%	0%	2%
Disjointness	97%	68%	35%	66%	3%	32%	65%	34%	0%	0%	0%	0%
Terms relation	81%	72%	53%	67%	17%	22%	47%	30%	2%	6%	0%	3%
Instance	67%	84%	60%	70%	30%	16%	35%	27%	4%	0%	5%	3%
Cardinality	84%	88%	64%	78%	16%	12%	36%	22%	0%	0%	0%	0%
Terms relation and cardinality	83%	55%	60%	58%	6%	25%	40%	33%	11%	45%	0%	10%
Hierarchy and disjointness	78%	80%	70%	75%	22%	20%	30%	25%	0%	0%	0%	0%
Terms relation and hierarchy	60%	60%	50%	53%	10%	40%	50%	47%	30%	0%	0%	0%
Terms relation and union	90%	30%	90%	70%	10%	70%	10%	30%	0%	0%	0%	0%



(a) It is useful

(b) It makes the things I want to accomplish easier to get done

Fig. 6. Summary of the usefulness results.

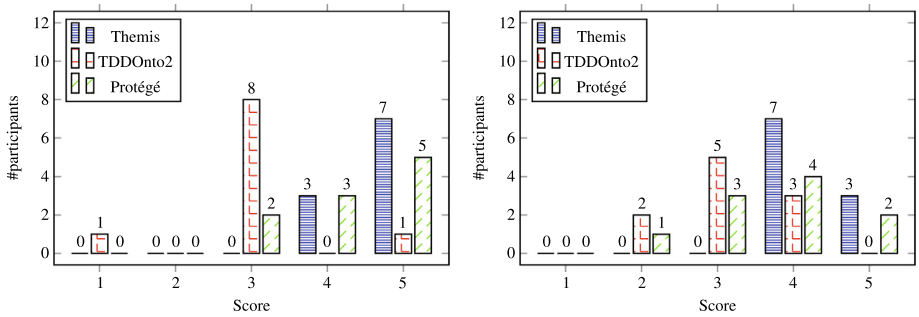
During the development of the CORAL corpus, significant difficulties to find available real-world requirements were found, which hindered the analysis of the requirements specifications and, consequently, limited the defined testing language. Publishing ontology requirements online would help the analysis of their specification and, therefore, would facilitate research related to their definition, formalisation, and verification.

In the evaluation analysis, Themis and TDDOnto2, both tools that use a testing language for the ontology verification process, were compared to Protégé, which is a tool that does not use a testing language. As shown in the empirical analysis (Section 4), both Themis and TDDOnto2 reduced the number of errors during the verification process for those users that are familiar with the OWL language, as well as for developers who are experts in it, compared to Protégé. Moreover, Protégé is the tool that had a stable time spent per requirement, but also the worst results with regard to errors made during the verification process.

From the results obtained during the evaluation of the testing process, it was concluded that users without any ontology background found it difficult to understand the restrictions that can be described in an ontology. Consequently, it is difficult for them to go beyond asking for the presence of classes and properties, which usually is not enough for the verification process. Moreover, ontology visualisation or examples of use are needed by them to understand the structure of the ontology. Furthermore, domain experts and users that have knowledge about ontologies, even if they are not experts, managed to verify an ontology using tests and to analyse the restrictions included in it. Therefore, it was concluded that ontology verification testing processes should consider ontology engineers and practitioners with a minimum of knowledge regarding ontologies as potential users since a manual review of the verification status and tests is needed.

During the experiment, it was also found that all types of participants made mistakes during the verification process, even those that are experts in the OWL language. These mistakes refer to incorrect answers, i.e., to affirm that a requirement is satisfied by an ontology when it is not or vice versa. Having incorrect answers during the verification activity can affect the development of the ontology, as it helps to ensure that all the expected requirements are satisfied. Therefore, this fact reinforces the need for testing approaches that ontology engineers should use to develop ontologies that satisfy their expected requirements.

This verification testing method was used as the basis in the conformance testing method proposed by Fernández-Izquierdo and García-Castro [13], in which the tests are used to check whether an ontology satisfies the requirements of a standard.



(a) I am satisfied with it

(b) It works the way I want it to work

Fig. 7. Summary of the satisfaction results.

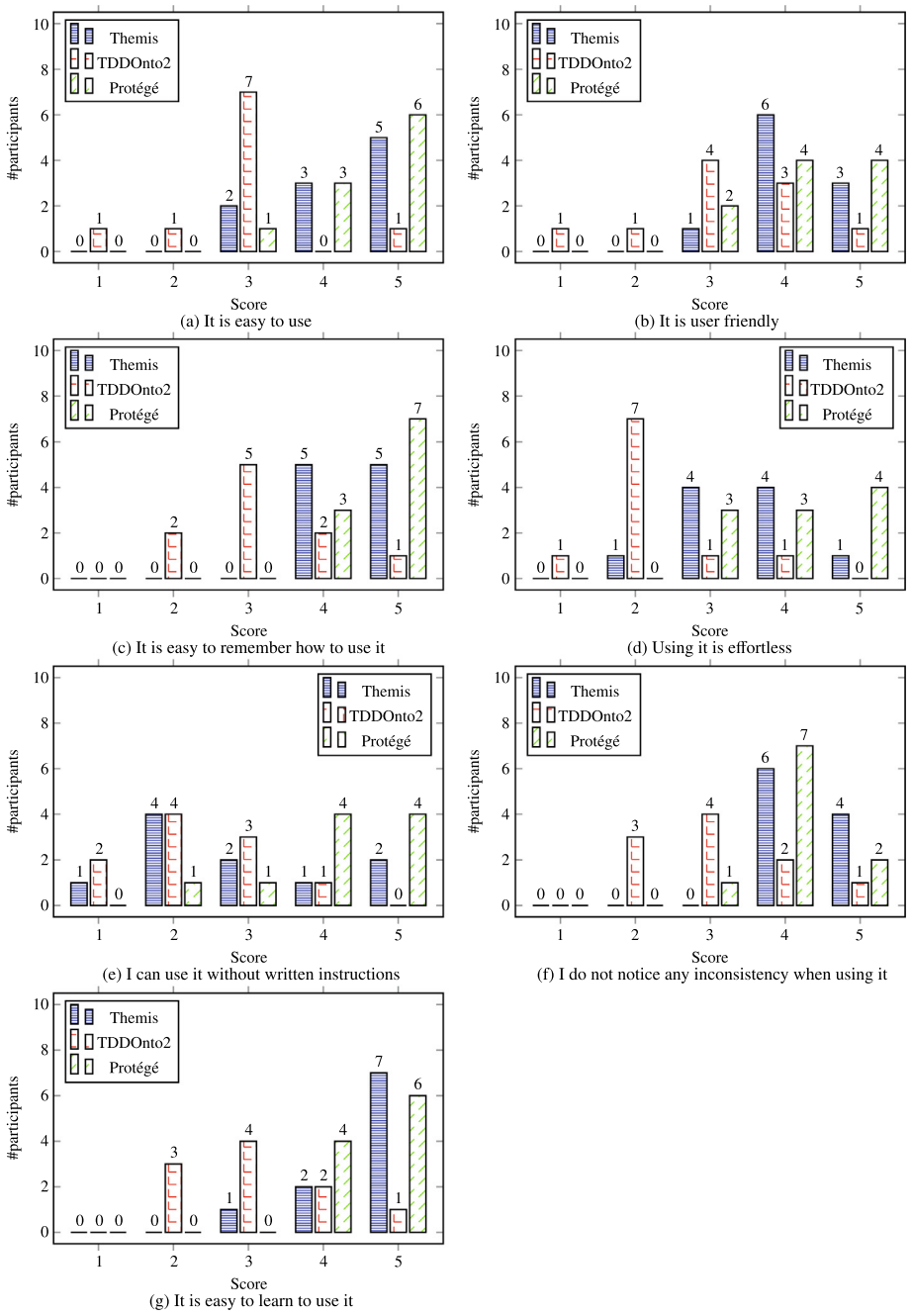


Fig. 8. Summary of the ease of use results.

Future work will be directed to an extension of the CORAL corpus with requirements written in other languages, e.g., in Spanish. This extension would improve the analysis of the requirements specification, leading to the analysis of more types of requirements and, consequently, to richer testing languages. Another line of future work is the implementation of a recommendation system for the translation of requirements into tests, intending to recommend potential tests based on requirements. This recommendation system could reduce the time spent on the definition of tests. Finally, future work will be also directed to the analysis of how the verification method can be used together with ontology evolution approaches (e.g., [7] or [11]). This analysis could also be used to determine how the terms defined in the requirements evolve alongside the ontology.

CRediT authorship contribution statement

Alba Fernandez-Izquierdo: Conceptualization, Methodology, Software. **Raul Garcia-Castro:** Supervision, Writing - review & editing.

Declaration of Competing Interest

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

Acknowledgements

This work is supported by a Predoctoral grant from the I+D+i program of the Universidad Politécnica de Madrid.

References

[1] G. Aguado De Cea, A. Gómez-Pérez, E. Montiel-Ponsoda, M.C. Suárez-Figueroa, Natural language-based approach for helping in the reuse of ontology design patterns, in: Proceedings of the 16th International Conference on Knowledge Engineering and Knowledge Management, Acitrezza, Italy, September 29–October 2, 2008, Springer, 2008, pp. 32–47.
[2] F. Baader, I. Horrocks, U. Sattler, Description logics, Foundations of Artificial Intelligence 3 (2008) 135–179. Elsevier..
[3] K. Beck, Test-driven development: by example, Addison-Wesley, 2003, Professional.
[4] C. Bezerra, F. Freitas, F. Santana, Evaluating ontologies with competency questions, in: Proceedings of the IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology, Atlanta, Georgia, USA, November 17–20, 2013, vol. 3, IEEE Computer Society, 2013, pp. 284–285..
[5] C. Bezerra, F. Santana, F. Freitas, CQChecker: A tool to check ontologies in OWL-DL using competency questions written in controlled natural language, Learning and Nonlinear Models 12 (2) (2014) 115–129. SBRN..
[6] E. Blomqvist, A.S. Sepour, V. Presutti, Ontology testing-methodology and tool, in: Proceedings of the 18th International Conference on Knowledge Engineering and Knowledge Management, Galway City, Ireland, October 8–12, 2012, Springer, 2012, pp. 216–226.
[7] D. Cavaliere, S. Senatore, OntoDrift: a Semantic Drift Gauge for Ontology Evolution Monitoring. In Proceedings of the 6th Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW) co-located with the 19th International Semantic Web Conference (ISWC 2020), Virtual event, November 1st, 2020, volume 2821 of CEUR Workshop Proceedings, 2020..
[8] E. Daga, E. Blomqvist, A. Gangemi, E. Montiel-Ponsoda, N. Nikitina, V. Presutti, B. Villazón-Terrazas, NeOn D2. 5.2 Pattern based ontology design: methodology and software support,Technical report. NeOn Project, 2010, http://www. neon-project. org..
[9] K. Davies, C.M. Keet, A. Ławrynowicz, TDDonto2: A Test-Driven Development Plugin for arbitrary TBox and ABox axioms, in: Proceedings of the 14th European Semantic Web Conference, Portoroz, Slovenia, May 28–June 1, 2017, Springer, 2017, pp. 120–125.
[10] M. Dennis, K. van Deemter, D. Dell'Aglio, J.Z. Pan, Computing authoring tests from competency questions: experimental validation, in: Proceedings of the 16th International Semantic Web Conference, Vienna, Austria, October 21–25, 2017, Springer, 2017, pp. 243–259.
[11] J.C. dos Reis, D. Dinh, M.D. Silveira, C. Pruski, C. Reynaud-Delaıtre, Recognizing lexical and semantic change patterns in evolving life science ontologies to inform mapping adaptation, Artificial Intelligence in Medicine 63 (3) (2015) 153–170.
[12] A. Fernández-Izquierdo, R. García-Castro, Requirements behaviour analysis for ontology testing, in: Proceedings of the 21st International Conference on Knowledge Engineering and Knowledge Management (EKAW 2018), Nancy, France, November 12–26, 2018, Springer, 2018, pp. 114–130..
[13] A. Fernández-Izquierdo, R. García-Castro, Conformance testing of ontologies through ontology requirements, Engineering Applications of Artificial Intelligence 97 (2021) 104026.
[14] A. Fernández-Izquierdo, M. Poveda-Villalón, R. García-Castro, CORAL: A corpus of ontological requirements annotated with lexico-syntactic patterns, in: Proceedings of 16th Extended Semantic Web Conference, Portorož, Slovenia, June 2–6, 2019, Springer, 2019, pp. 443–458.
[15] A. Fernández-Izquierdo, R. García-Castro, Themis: A tool for validating ontologies, in: Proceedings of the 31st International Conference on Software Engineering and Knowledge Engineering (SEKE 2019), Lisbon, Portugal, July 10–12, 2019, 2019.
[16] A. Gangemi, V. Presutti, Ontology design patterns, Handbook on Ontologies, Springer, 2009, pp. 221–243.
[17] S. García-Ramos, A. Otero, M. Fernández-López, OntologyTest: A tool to evaluate ontologies through tests defined by the user, in: Proceedings of the 10th International Work-Conference on Artificial Neural Networks on Artificial Neural Networks, Salamanca, Spain, June 10–12, 2009, Springer, 2009, pp. 91–98.
[18] M. Grüninger, M.S. Fox, Methodology for the design and evaluation of ontologies, in: Proceedings of the Workshop on Basic Ontological Issues in Knowledge Sharing, 1995.
[19] L. Halilaj, N. Petersen, I. Grangel-González, C. Lange, S. Auer, G. Coskun, S. Lohmann, Vocol: An integrated environment to support version-controlled vocabulary development, in: Proceedings of the 20th International Conference on Knowledge Engineering and Knowledge Management, Bologna, Italy, November 19–23, 2016, Springer, 2016, pp. 303–319.
[20] C.M. Keet, A. Ławrynowicz, Test-Driven Development of ontologies, in: Proceedings of 13th European Semantic Web Conference, Heraklion, Crete, Greece, May 29–June 2, 2016, Springer, 2016, pp. 642–657.
[21] W. Kirch (Ed.), Pearson's Correlation Coefficient, Springer, Netherlands, 2008.
[22] I. Kollia, B. Glimm, I. Horrocks, SPARQL query answering over OWL ontologies, in: Proceedings of 8th Extended Semantic Web Conference, Heraklion, Crete, Greece, May 29–June 2, 2011, Springer, 2011, pp. 382–396.
[23] A. Ławrynowicz, C.M. Keet, The TDDonto Tool for Test-Driven Development of DL Knowledge bases, in: Cape Town, South Africa (Eds.), Proceedings of the 29th International Workshop on Description Logics, Cape Town, South Africa, April 22-25, 2016, volume 1577. CEUR Workshop series, 2016.
[24] I. Lazar, S. Motogna, B. Pârv, Behaviour-Driven Development of Foundational UML Components, Electronic Notes in Theoretical Computer Science 264 (1) (2010) 91–105.
[25] A.M. Lund, Measuring usability with the use questionnaire, Usability Interface 8 (2) (2001) 3–6.
[26] E. Montiel-Ponsoda, Multilingualism in Ontologies - Building Patterns and Representation Models, LAP Lambert Academic Publishing, 2011.
[27] J. Pérez, M. Arenas, C. Gutierrez, Semantics and Complexity of SPARQL, ACM Transactions on Database Systems 34 (3) (2009) 16:1–16:45, ACM.
[28] S. Peroni, A simplified agile methodology for ontology development, in: Proceedings of the 13th OWL: Experiences and Directions Workshop and 5th OWL reasoner evaluation workshop, Springer, 2016, pp. 55–69.
[29] Y. Ren, A. Parvizi, C. Mellish, J.Z. Pan, K. Van Deemter, R. Stevens, Towards competency question-driven ontology authoring, in: Proceedings of the 11th European Semantic Web Conference, Crete, Greece, May 25–29, 2014, Springer, 2014, pp. 752–767.
[30] M.C. Suárez-Figueroa, G. Aguado de Cea, A. Gómez-Pérez, Lights and shadows in creating a glossary about ontology engineering, Terminology 19 (2) (2013) 202–236. John Benjamins Publishing Company..
[31] M.C. Suárez-Figueroa, A. Gómez-Pérez, M. Fernández-López, The NeOn Methodology framework: A scenario-based methodology for ontology development, Applied Ontology 10 (2) (2015) 107–145. IOS Press..

[32] M.C. Suárez-Figueroa, A. Gómez-Pérez, B. Villazón-Terrazas, How to write and use the ontology requirements specification document, in: Proceedings of the International Conference on On the Move to Meaningful Internet Systems, Springer, 2009, pp. 966–982.
[33] M. Utting, B. Legeard, Practical Model-based Testing: A Tools Approach, Elsevier, 2010.
[34] D. Wiśniewski, J. Potoniec, A. Ławrynowicz, C.M. Keet, Analysis of ontology competency questions and their formalizations in SPARQL-OWL, Journal of Web Semantics 59 (2019) 100534. Elsevier..
[35] L. Zemmouchi-Ghomari, A.R. Ghomari, Translating natural language competency questions into SPARQLQueries: a case study, in: Proceedings of the First International Conference on Building and Exploring Web Based Environments, Seville, Spain, January 27–February 1, 2013, IARIA XPS Press, 2013, pp. 81–86.