UNIVERSITY OF CAPE TOWN
IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD

Department of Computer Science

Computer Science Honours Final Paper 2016

Title: Towards Test-Driven Development of Ontologies: An Analysis of Testing Algorithms

Author: Kieren Davies

Project abbreviation: TDDON

Supervisor: Maria Keet

Category	**Min**	**Max**	**Chosen**
Requirement Analysis and Design	0	20	-
Theoretical Analysis	0	25	25
Experiment Design and Execution	0	20	-
System Development and Implementation	0	15	-
Results, Findings and Conclusion	10	20	20
Aim Formulation and Background Work	10	15	15
Quality of Paper Writing and Presentation	10		10
Quality of Deliverables	10		10
Overall General Project Evaluation	0	10	-
Total marks			80

Towards Test-Driven Development of Ontologies: An Analysis of Testing Algorithms

Kieren Davies
University of Cape Town
kieren@kdavi.es

ABSTRACT

Test-Driven Development is a software engineering methodology with proposed application to ontology engineering [9]. It requires sophisticated tools, but all existing ontology testing tools have shortcomings which limit their usefulness, and no formal analysis has been done of their algorithms. In order to facilitate the development and advancement of such tools, we present a formal model of ontology testing and new testing algorithms. We rigorously prove correctness of these algorithms.

1. INTRODUCTION

Ontologies, and ontology engineering, have become increasingly relevant in the past decade. They are regarded as a critical component of the Semantic Web [2], and have been employed successfully in fields ranging from genetics [4] to news and broadcasting [15].

Despite this, ontologies have not seen widespread adoption within business and industry [7]. We postulate that one of the contributing factors is the state of ontology engineering methodologies, which lag behind software engineering methodologies in terms of both maturity and adoption [6, 17]. In particular, there are no published methodologies which explicitly incorporate automated testing, which has become a staple of software engineering.

There exist some tools for testing ontologies [10, 19, 16], but they all share two notable shortcomings: certain axioms cannot be tested even though they are permitted in OWL 2 [11], such as $\forall R.C \sqsubseteq D$, and test results are either "pass" or "fail" with no further information as to the nature of failure. Furthermore, there has been no rigorous analysis of the techniques and algorithms used for ontology testing. In this paper we present algorithms to test axioms against ontologies and prove their correctness. The algorithms provide new functionality to address the aforementioned shortcomings.

We aim for the algorithms to each handle the general case of an axiom, and avoid having multiple algorithms to handle variants of a single kind of axiom.

We aim for all algorithms to be acceptably fast. Specifically, we do not permit reclassification of an ontology in any test, because on typical hardware this commonly takes minutes or hours, which we deem unacceptably slow.

Lastly we require the algorithms to support operation with any reasoner compliant with OWL 2 semantics.

In section 2 we justify why testing is applicable to ontologies. In section 3 we examine prior work that has been done on this topic, and the shortcomings in the state of the art. In section 4 we consider how ontology reasoners may be applied to develop testing algorithms. In section 5 we present a formal model of testing, and in section 6 we employ the model to describe and analyse testing algorithms. In section 7 we demonstrate the use cases of these algorithms with examples. In section 8 we discuss the usefulness of the new algorithms and how they may be implemented in new or existing tools. Lastly, in section 9 we conclude and identify scope for future work.

2. RATIONALE FOR TESTING

In software engineering, *Test-Driven Development* (TDD) [1] is a methodology based on two rules:

- Write new code only if an automated test has failed.
- Eliminate duplication.

This induces a "red–green–refactor" pattern of development: first write a new test which fails, then write code which makes it pass with minimal effort, then remove resultant duplication and restructure if necessary. In this way, tests serve to define desired functionality. The process is usually facilitated with a test harness which runs tests automatically and generates reports.

TDD has been shown to improve code quality [14], especially in complex projects, and it is also believed to improve productivity and morale [1]. In light of this, it has been proposed that TDD should be incorporated into new or existing ontology development methodologies [9].

Ontologies, like computer programs, can become complex to the point that it is difficult for a human author to predict the consequences of changes, especially if the author is inexperienced. Automated tests are therefore useful to detect unintended consequences. As an illustrative example, suppose an author creates the following classes:

$$\texttt{Giraffe} \sqsubseteq \texttt{Herbivore} \sqsubseteq \texttt{Mammal} \sqsubseteq \texttt{Animal}$$

The author then realises that not all herbivores are mammals, so changes `Herbivore` to be a subclass only of `Animal`. But now `Giraffe` is no longer a derived subclass of `Mammal`,

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

and an application which uses this ontology to enumerate mammals would erroneously exclude giraffes. This mistake could be caught by a simple test which declares that `Giraffe` should be a subclass of `Mammal`.

Superficially, it seems like this problem can be solved by not writing tests for axioms but instead just adding those axioms directly to the ontology. However, adding such axioms introduces redundancy, making modification of the ontology more difficult, and in some circumstances increasing the complexity of reasoning [18]. Adding only a test instead ensures correctness without bloating the ontology.

Tests may also be used outside of an automated test suite to explore and understand an ontology. For example, an author might be assessing an ontology of animals for reuse and wants to verify that $\texttt{Giraffe} \sqsubseteq \texttt{Mammal}$. The author can simply create a corresponding temporary test and observe the result. This saves the time it would take to browse the inferred class hierarchy in a development environment such as Protégé.

A similar approach can be employed when developing a new ontology. Before adding a new axiom, the author can create a temporary test to determine if the axiom is already entailed, if it will result in a contradiction or unsatisfiable class, or if it can be safely added. The standard approach of adding an axiom and then observing the consequences involves reclassification, which is typically very slow, and which a test of a single axiom can avoid.

This gives us two broad use cases:

1. Declare many tests alongside an ontology and evaluate them all together in order to demonstrate quality or detect regressions.
2. Evaluate temporary tests as needed in order to explore an ontology or predict the consequences of adding a new axiom.

To satisfy both of these, tests must not reclassify the ontology, and they must produce results which identify the consequences of adding an axiom.

3. RELATED WORK

To the best of our knowledge, there are only three existing tools which implement ontology testing.

TDDonto [10] is a Protégé plugin which allows test axioms to be specified in Protégé's expression syntax. Several implementation strategies were tried, and in evaluation it was found that directly querying the reasoner through the OWL API [12] is faster than evaluating SPARQL queries with OWL-BGP [13] or introducing "mock" individual assertions. It supports testing only certain object property axioms, and it does not support data properties.

Tawny-OWL [19] is an ontology development framework implemented in Clojure. It provides predicate functions which query the reasoner which can be used in conjunction with any testing framework, such as the built-in "clojure.test". It has the disadvantage of being totally seperated from the usual ontology development tools and requiring tests to be written in Clojure, which is not a widely known and used language. It does not support querying object or data properties.

SCONE [16] is a tool based on Cucumber [3] which evaluates tests written in controlled natural language in a procedural style. It encourages the pattern of creating mock individuals and making assertions and inferences on them. It is also separated from the usual ontology development tools. It does not support testing object or data properties.

None of these three support the full range of axioms permitted in OWL 2. Most notably, in none of them is it possible to directly test axioms of the form $C \sqsubseteq D$ where C is not a named class, such as $\forall R.E \sqsubseteq D$.

Additionally, all three of these tools give only limited information about the result of any test. Tawny-OWL and SCONE report only pass or fail; TDDonto further reports if any entity in the axiom under test is not declared in the ontology. This hinders their usefulness as a means to explore an ontology or aid in development.

Furthermore, no attempt has been made to rigorously prove the correctness of the testing algorithms used by any of these tools.

There is clear scope for these shortcomings to be addressed by new algorithms and tools which provide full coverage of OWL 2 axioms and which return detailed test results, as well as proofs of their correctness.

4. APPLICATION OF REASONERS

There are numerous automated reasoners which are compliant with OWL 2, most notably HermiT [5], with several mechanisms for making use of them. OWL API [12] is a Java library which defines a widely-supported standard interface for reasoners. OWL-BGP [13] is a Java library which implements SPARQL, an ontology query language.

As mentioned in section 3, evaluation of TDDonto identified that OWL-BGP introduces an efficiency overhead which is not present in OWL API [10]. Therefore we focus only on OWL API and the functionality it provides.

Evaluation of TDDonto also found that implementations which introduced temporary "mock" individuals were substantially slower than all others. The cause was not explicitly identified, but it is likely due to the need for reclassification of the ontology to include the new assertions. As stipulated in section 1, we do not permit reclassification and therefore we cannot introduce new axioms in our testing algorithms. Instead, after the ontology is initially classified, we only make queries, which are assumed to be acceptably efficient.

The OWL API reasoner interface specifies a "convenience" method named ISENTAILED which accepts any axiom and returns a boolean indicating whether or not that axiom is entailed. However, it is not mandatory for reasoners to implement this method. We therefore do not permit its use.

In order to make use of reasoner methods and perform rigorous analysis on the algorithms in which they are used, we require a formalisation of available methods and their returned values. We begin with a prerequisite definition.

Definition 1. The *signature* $\Sigma(A)$ of an axiom or ontology A, the set of all symbols in its formulae. The *class signature* $\Sigma_C(A)$ and *individual signature* $\Sigma_I(A)$ are the subsets of $\Sigma(A)$ which respectively contain only classes and individuals.

Now we identify relevant methods and their returned values. Let O be the ontology under test. Here and in the remainder of the paper we use the variables C, D, and E for class expressions; N for a named class; a and b for individuals; and R for an object property expression.

Definition 2. The following methods are available from

the reasoner.

$$\textsc{isSatisfiable}(C) \iff O \nvdash C \sqsubseteq \bot$$
$$\textsc{getSubClasses}(C) = \{N \in \Sigma_C(O) \mid O \vdash N \sqsubseteq C\}$$
$$\textsc{getInstances}(C) = \{a \in \Sigma_I(O) \mid O \vdash a : C\}$$
$$\textsc{getTypes}(a) = \{N \in \Sigma_C(O) \mid O \vdash a : N\}$$
$$\textsc{getSameIndividuals}(a) = \{b \in \Sigma_I(O) \mid O \vdash b \equiv a\}$$
$$\textsc{getDifferentIndividuals}(a) = \{b \in \Sigma_I(O) \mid O \vdash b \not\equiv a\}$$

Note that getSubClasses returns the union of equivalent classes and strict subclasses.

We make the assumption that reasoners implement these methods correctly and efficiently. No thorough low-level analysis of any reasoners has been done, but the high-level techniques used by the most popular reasoners, such as HermiT's hypertableau calculus [5], are widely accepted to be correct. In addition, the popular reasoners have been subject to extensive practical testing.

5. A MODEL OF TESTING

In order to rigorously examine any testing algorithms, we need a formal description of what it means to test an axiom against an ontology. In line with the uses cases identified in section 2, we define the possible test results. They are listed in order from most grave failure to least.

Ontology already inconsistent: The ontology is inconsistent before considering the axiom under test; that is, it contains a contradiction. The reasoner cannot meaningfully respond to queries, so no claims can be made about the axiom.

Ontology already incoherent: The ontology is incoherent before considering the axiom; that is, one or more of its named classes are unsatisfiable. This confounds certain methods which will be used to evaluate tests.

Missing entity in axiom: The axiom contains one or more named classes or properties which are not declared in the ontology. This result is probably caused by a mistake in the test declaration, so it is distinguished from the results below.

Axiom causes inconsistency: If the axiom were to be added to the ontology, it would cause it to become inconsistent.

Axiom causes incoherence: If the axiom were to be added to the ontology, it would cause one or more named classes to become unsatisfiable.

Axiom absent: The axiom is not entailed by the ontology, but it could be added without negative consequences.

Axiom entailed: The axiom is already entailed by the ontology.

Viewed in the context of TDD, only the final result in this list, "axiom entailed", should be considered a pass; all others are test failures.

If the ontology is already inconsistent or incoherent, every test will produce the same result. In this way, these two cases apply to the entire suite of tests rather than to any one, and so they should be checked only once as preconditions before evaluating any tests. Therefore we do not consider them in the formal definition of a test result, or in any of the algorithms.

Similarly, we do not directly address missing entities, as this can be a simple check performed at the start of each test which does not affect how it is otherwise evaluated.

Since there is no ambiguity, we henceforth abbreviate the remaining cases to "inconsistent", "incoherent", "absent", and "entailed".

This leads to the following formalisation.

Definition 3. Given an ontology O which is consistent and coherent, and an axiom A such that $\Sigma(A) \subseteq \Sigma(O)$, the result of testing A against O is

$$\text{test}_O(A) = \begin{cases} \text{entailed} & \text{if } O \vdash A \\ \text{inconsistent} & \text{if } O \cup A \vdash \bot \\ \text{incoherent} & \text{if } O \cup A \nvdash \bot \\ & \quad \wedge\, (\exists C \in \Sigma_C(O)) \\ & \qquad O \cup A \vdash C \sqsubseteq \bot \\ \text{absent} & \text{otherwise} \end{cases}$$

The resultant values are ordered according to graveness of failure.

$$\text{entailed} < \text{absent} < \text{incoherent} < \text{inconsistent}$$

6. ALGORITHMS AND ANALYSIS

We now present the algorithms and analysis, in the context of an ontology O. We base the coverage on OWL 2.

We exclude entity declarations and datatype definitions because they cannot meaningfully be tested. We exclude object property axioms, as well as object property assertions, due to the difficulty if identifying inconsistencies, which has been addressed by prior work [8]. However we include object property domain and range axioms, and functional and reflexive and their inverses, as these can be expressed as class axioms. We exclude data property and annotation axioms and data property assertions, which would be redundant because their grammar and semantics are equivalent to a fragment of object properties. We also exclude HasKey axioms. This leaves four class axioms, three assertions, and six object property axioms which will be covered.

As justified in section 5, we assume as preconditions that the ontology under test O is consistent and coherent, and the axiom under test contains only entities which are declared in the ontology.

Each algorithm is named according to the axiom it tests, as written in OWL 2 functional syntax, prepended with "test". For example, the algorithm for testing SubClassOf axioms is named testSubClassOf.

We address the class axioms in section 6.1, assertions in section 6.2, and object property axioms in section 6.3.

6.1 Class axioms

In the class axioms permitted by OWL 2, all arguments may be arbitrary class expressions, not only named classes, except DisjointUnion$(N, C_1, \ldots, C_n)$ in which N must be a named class. Consequently, to determine if $C \sqsubseteq D$ it is not sufficient to check if $C \in \textsc{getSubClasses}(D)$, because C will not occur in this set if it is not a named class. To resolve this, we build class expressions from the arguments and query them for satisfiability and instances.

Algorithm 1 tests subsumption of class expressions.

Algorithm 1 test $C \sqsubseteq D$

Input: C, D class expressions

```
1: function testSubClassOf(C, D)
2:     if getInstances(C ⊓ ¬D) ≠ ∅ then
3:         return inconsistent
4:     else if getSubClasses(C ⊓ ¬D) ≠ ∅ then
5:         return incoherent
6:     else if isSatisfiable(C ⊓ ¬D) then
7:         return absent
8:     else
9:         return entailed
10:    end if
11: end function
```

Lemma 1. *For any set of axioms O and class expressions C and D,*

$$O \vdash C \sqsubseteq D \iff O \vdash C \sqcap \neg D \sqsubseteq \bot$$

Proof. For every interpretation $\mathcal{I} \vDash O$,

$$\begin{aligned} & O \vdash C \sqsubseteq D \\ \iff & \mathcal{I} \vDash C \sqsubseteq D \\ \iff & (C \sqcap \neg D)^{\mathcal{I}} = \emptyset = \bot^{\mathcal{I}} \\ \iff & \mathcal{I} \vDash C \sqcap \neg D \sqsubseteq \bot \\ \iff & O \vdash C \sqcap \neg D \sqsubseteq \bot \end{aligned}$$

The first and last bi-implications hold because we consider every possible interpretation. □

Proposition 1. testSubClassOf *is sound with respect to entailment. That is,*

$$\text{testSubClassOf}(C, D) = \text{entailed} \implies \text{test}_O(C \sqsubseteq D) = \text{entailed}$$

Proof. The algorithm can only return entailed at line 9, so the three if-conditions must all be false. So

$$\begin{aligned} & \text{getInstances}(C \sqcap \neg D) = \emptyset \\ & \quad \wedge \text{getSubClasses}(C \sqcap \neg D) = \emptyset \\ & \quad \wedge \neg \text{isSatisfiable}(C \sqcap \neg D) \end{aligned} \quad (1)$$

Now suppose $O \nvdash C \sqsubseteq D$. By lemma 1, $O \nvdash C \sqcap \neg D \sqsubseteq \bot$. In other words, $C \sqcap \neg D$ is satisfiable, which contradicts the last term of equation 1. Hence the supposition is false, so

$$\begin{aligned} & O \vdash C \sqsubseteq D \\ \iff & \text{test}_O(C \sqsubseteq D) = \text{entailed} \end{aligned}$$ □

Proposition 2. testSubClassOf *is complete with respect to entailment. That is,*

$$\text{test}_O(C \sqsubseteq D) = \text{entailed} \implies \text{testSubClassOf}(C, D) = \text{entailed}$$

Proof. As indicated in the proof of proposition 1, the algorithm returns entailed if equation 1 holds.

From $\text{test}_O(C \sqsubseteq D) = \text{entailed}$ we have that $O \vdash C \sqsubseteq D$, and by lemma 1, $O \vdash C \sqcap \neg D \sqsubseteq \bot$ so the last term of the equation is true.

Since $C \sqcap \neg D$ is unsatisfiable, by the coherence precondition it has no named subclasses, and by the consistency precondition it has no instances. Therefore the first and second terms of the equation are also true.

Therefore equation 1 holds, and so the algorithm returns entailed. □

Proposition 3. testSubClassOf *is sound with respect to inconsistency.*

Proof. The algorithm can only return inconsistent at line 3, so the first if-condition holds, so

$$\text{getInstances}(C \sqcap \neg D) \neq \emptyset$$

which means there exists an individual a such that

$$\begin{aligned} & a : C \sqcap \neg D \\ \iff & a : C \ \wedge \ a : \neg D \end{aligned}$$

Under $O \cup (C \sqsubseteq D)$ it follows also that $a : D$, which is a contradiction, so

$$\begin{aligned} & O \cup (C \sqsubseteq D) \vdash \bot \\ \iff & \text{test}_O(C \sqsubseteq D) = \text{inconsistent} \end{aligned}$$ □

Proposition 4. testSubClassOf *is complete with respect to inconsistency.*

Proof. We have that O is consistent, so it has an interpretation, but $O \cup (C \sqsubseteq D)$ is inconsistent, so it has no interpretations.

Suppose the algorithm does not return inconsistent. Then it must be that

$$\text{getInstances}(C \sqcap \neg D) = \emptyset$$

But in this case there exists an interpretation $\mathcal{I}$ which models both O and $O \cup (C \sqsubseteq D)$. Let $\mathcal{I}$ be the interpretation of O with the smallest domain. This means that the interpretation of any class E only contains elements which correspond to individuals which must be in that class.

$$E^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid (\exists a \in \Sigma_I(O))\ a : E \wedge a^{\mathcal{I}} = x\}$$

This clearly still models O because every individual is still in all classes it is entailed to be in.

Under the supposition, we have that

$$(C \sqcap \neg D)^{\mathcal{I}} = \emptyset$$

So for any individual a,

$$a : \neg(C \sqcap \neg D)$$

Letting $a : C$,

$$\begin{aligned} a : \neg(C \sqcap \neg D) & \implies a : \neg C \sqcup D \\ & \implies a : D \end{aligned}$$

From the construction of $\mathcal{I}$, this means that

$$\begin{aligned} & C^{\mathcal{I}} \subseteq D^{\mathcal{I}} \\ \implies & \mathcal{I} \vDash C \sqsubseteq D \end{aligned}$$

So $\mathcal{I}$ also models $O \cup (C \sqsubseteq D)$.

This contradicts the initial condition that $O \cup (C \sqsubseteq D)$ is inconsistent, so the supposition must be false, and therefore the algorithm returns inconsistent. □

Proposition 5. testSubClassOf *is sound with respect to incoherence.*

PROOF. The algorithm can only return incoherent at line 5, so the first if-condition must be false and the second true. So

$$\text{getInstances}(C \sqcap \neg D) = \emptyset \\ \wedge \text{getSubClasses}(C \sqcap \neg D) \neq \emptyset$$

Therefore, by the second term, there exists some named class $N \in \Sigma_C(O)$ such that

$$N \sqsubseteq C \sqcap \neg D$$

By the contrapositive of proposition 4, $O \cup (C \sqsubseteq D)$ is consistent, so by lemma 1,

$$\begin{aligned} & O \cup (C \sqsubseteq D) \vdash C \sqcap \neg D \sqsubseteq \bot \\ \implies & O \cup (C \sqsubseteq D) \vdash N \sqsubseteq \bot \\ \implies & \text{test}_O(C \sqsubseteq D) = \text{incoherent} \quad \square \end{aligned}$$

PROPOSITION 6. testSubClassOf *is complete with respect to incoherence.*

PROOF. If the axiom $C \sqsubseteq D$ is added to O, then the only classes which are affected are C and its subclasses. Consider a named class $N \sqsubseteq C$. If $O \nvdash N \sqsubseteq \neg D$ then it is possible that any element in the $N^{\mathcal{I}}$ is also in $D^{\mathcal{I}}$ and thus it is possible that $N \sqsubseteq D$. If this is true for all such N, then they are all satisfiable in $O \cup (C \sqsubseteq D)$ which is therefore coherent, so it must not be true for at least one N. That is,

$$\begin{aligned} & (\exists N \in \Sigma_C(O)) \ N \sqsubseteq C \wedge N \sqsubseteq \neg D \\ \iff & (\exists N \in \Sigma_C(O)) \ N \sqsubseteq C \sqcap \neg D \\ \iff & (\exists N \in \Sigma_C(O)) \ N \in \text{getSubClasses}(C \sqcap \neg D) \\ \iff & \text{getSubClasses}(C \sqcap \neg D) \neq \emptyset \end{aligned}$$

From the contrapositive of proposition 4 we have that the first if-condition is false, and we have shown that the second if-condition is true, so the algorithm returns incoherent. $\square$

THEOREM 1. testSubClassOf *is correct and terminating.*

PROOF. It has been shown that the algorithm is sound and complete with respect to entailment, inconsistency, and incoherence. By definition, the result is absent when it is not one of these other three. Therefore the algorithm returns the correct result in all cases.

Termination is trivial, since the algorithm contains no loops or recursion. $\square$

Algorithm 2 tests equivalence if class expressions. In the algorithm we use the integer variables i and j to iterate over the indices of the class expressions given as arguments, and the variables r and r' to store intermediate test results.

Algorithm 2 test $C_1 \equiv \ldots \equiv C_n$

Input: $C_1, \ldots, C_n$ class expressions
$n \geq 2$

```
1: function testEquivalentClasses(C_1, ..., C_n)
2:     r ← entailed
3:     for i ← 1 to n do
4:         for j ← 1 to n do
5:             r' ← testSubClassOf(C_i, C_j)
6:             r ← max(r, r')
7:         end for
8:     end for
9:     return r
10: end function
```

THEOREM 2. testEquivalentClasses *is correct and terminating.*

PROOF. By definition, $C_i \equiv C_j$ if and only if $C_i \sqsubseteq C_j$ and $C_j \sqsubseteq C_i$. It is immediately clear from this that the algorithm is sound with respect to inconsistency and incoherence, and sound and complete with respect to entailment. It remains to be shown that it is complete with respect to inconsistency and incoherence.

Suppose the algorithm does not return inconsistent. Then, for all $i, j \leq n$

$$\text{testSubClassOf}(C_i, C_j) \neq \text{inconsistent}$$

otherwise r would take the value inconsistent because it is greater than other values. Therefore, for any individual a,

$$a : C_i \implies a : C_j$$

So if a is in any of $C_1, \ldots, C_n$ it is in all of them. By the same construction as in the proof of proposition 4, $O \cup (C_1 \equiv \ldots \equiv C_n)$ has an interpretation and is not inconsistent. Therefore, by contrapositive, the algorithm is complete with respect to inconsistency.

Suppose the algorithm does not return incoherent. If it returns inconsistent then by the above $O \cup (C_1 \equiv \ldots \equiv C_n)$ is inconsistent. If not, then it is consistent, and for all $i, j \leq n$

$$\text{getSubClasses}(C_i \sqcap \neg C_j) = \emptyset$$

As in the proof of proposition 6, there exists an interpretation $I \vDash O$ where, for any class expression E,

$$E^{\mathcal{I}} \neq \emptyset \iff (\exists N \in \Sigma_C(O)) \ N \sqsubseteq E$$

So again under $O \cup (C_1 \equiv \ldots \equiv C_n)$ we have

$$\begin{aligned} & (C_i \sqcap \neg C_j)^{\mathcal{I}} = \emptyset \\ \implies & C_i^{\mathcal{I}} = (C_i \sqcap C_j)^{\mathcal{I}} \\ \implies & \mathcal{I} \vDash C_i \sqsubseteq C_j \end{aligned}$$

This is true for all pairs, so

$$\mathcal{I} \vDash C_1 \equiv \ldots \equiv C_n$$

So $\mathcal{I}$ also models $O \cup (C_1 \equiv \ldots \equiv C_n)$ which is therefore coherent. Therefore, by contrapositive, the algorithm is complete with respect to incoherence.

All loops are bounded by n, which is finite, and all calls to testSubClassOf terminate, therefore the entire algorithm terminates. $\square$

It is not sufficient to implement testEquivalentClasses by simply testing each of C_2 to C_n for equivalence to C_1, as shown by a counterexample. Let

$$O = \{N \equiv C_2 \sqcap \neg C_3\}$$

It can be seen that $\text{testSubClassOf}(C_2, C_3)$ returns incoherent, but this would not be reporting by testing only $C_1 \sqsubseteq C_2$, $C_2 \sqsubseteq C_1$, $C_1 \sqsubseteq C_3$, and $C_3 \sqsubseteq C_1$.

Algorithm 3 tests pairwise disjointness of class expressions.

Algorithm 3 test DisjointClasses$(C_1, \ldots, C_n)$

Input: $C_1, \ldots, C_n$ class expressions
$n \geq 2$

```
1: function testDisjointClasses(C_1, ..., C_n)
2:     r ← entailed
3:     for i ← 1 to n − 1 do
4:         for j ← i + 1 to n do
5:             r' ← testSubClassOf(C_i, ¬C_j)
6:             r ← max(r, r')
7:         end for
8:     end for
9:     return r
10: end function
```

Proposition 7. testDisjointClasses *is sound and complete with respect to entailment.*

Proof. The algorithm returns entailed if and only if the call to testSubClassOf at line 5 return entailed for every iteration of the enclosing. If it did not, then r' would have a greater value which would then be bound to r and eventually returned. Formally, for all C_i and C_j in $\{C_1, \ldots, C_n\}$ with $i < j$,

$$\begin{aligned} &\textsc{testSubClassOf}(C_i, \neg C_j) = \text{entailed} \\ \iff &\textsc{getInstances}(C_i \sqcap C_j) \neq \emptyset \\ &\quad \wedge \textsc{getSubClasses}(C_i \sqcap C_j) \neq \emptyset \\ &\quad \wedge \neg\textsc{isSatisfiable}(C_i \sqcap C_j) \end{aligned}$$

If this is true then $C_i \sqcap C_j \sqsubseteq \bot$, which by definition makes C_i and C_j disjoint, so the axiom is entailed. Conversely, if the axiom is entailed then $C_i \sqcap C_j \sqsubseteq \bot$, and with the preconditions of consistency and coherence we have that all three terms are true and thus the algorithm returns entailed. □

Proposition 8. testDisjointClasses *is sound with respect to inconsistency.*

Proof. The algorithm returns inconsistent if and only if r' takes the value inconsistent for some iteration of the nested for-loops, since it is the greatest test result value. So for some C_i and C_j with $i < j$

$$\begin{aligned} &\textsc{testSubClassOf}(C_i, \neg C_j) = \text{inconsistent} \\ \iff &\textsc{getInstances}(C_i \sqcap C_j) \neq \emptyset \\ \iff &(\exists a \in \Sigma_I(O))\ \ a : C_i \sqcap C_j \end{aligned}$$

But with the axiom added to the ontology, $C_i \sqcap C_j \sqsubseteq \bot$, thus $a : \bot$ which is a contradiction, therefore

$$\text{test}_O(\text{DisjointClasses}(C_1, \ldots, C_n)) = \text{inconsistent} \quad \square$$

Proposition 9. testDisjointClasses *is complete with respect to inconsistency.*

Proof. Suppose the algorithm does not return inconsistent. Then for all C_i and C_j with $i < j$

$$\begin{aligned} &\textsc{testSubClassOf}(C_i, \neg C_j) \neq \text{inconsistent} \\ \iff &\textsc{getInstances}(C_i \sqcap C_j) = \emptyset \end{aligned}$$

Since C_i and C_j being disjoint means only that $C_i \sqcap C_j$ is empty, and it does not have any instances, the new axiom does not contradict O. Therefore O with the axiom added has interpretations, such as the interpretation of O with the smallest domain, and consequently it is consistent. Thus, by contrapositive, the proposition holds. □

Proposition 10. testDisjointClasses *is sound with respect to incoherence.*

Proof. If the algorithm returns incoherent then, by the contrapositive of proposition 9, for all C_i and C_j with $i < j$

$$\textsc{testSubClassOf}(C_i, \neg C_j) \neq \text{inconsistent}$$

But for some C_i and C_j it must be that

$$\begin{aligned} &\textsc{testSubClassOf}(C_i, \neg C_j) = \text{incoherent} \\ \iff &\textsc{getInstances}(C_i \sqcap C_j) = \emptyset \\ &\quad \wedge \textsc{getSubClasses}(C_i \sqcap C_j) \neq \emptyset \end{aligned}$$

From the last term it follows that there exists a named class $N \sqsubseteq C_i \sqcap C_j$. But if the axiom is added to O, this class is empty, resulting in incoherence. □

Proposition 11. testDisjointClasses *is complete with respect to incoherence.*

Proof. If DisjointClasses$(C_1, \ldots, C_n)$ is added to O, the only classes affected are the intersections $C_i \sqcap C_j$ and their subclasses, which become unsatisfiable. If this results in incoherence, then there must have been a named class $N \sqsubseteq C_i \sqcap C_j$ for some arguments C_i and C_j. If any N were not a subclass of such an intersection, then it would remain satisfiable with the addition of the axiom.

Note that, by the consistency precondition, no $C_i \sqcap C_j$ has instances.

Given this C_i and C_j, without loss of generality let $i < j$. When the for-loops reach the respective i and j, we will have that

$$\begin{aligned} &\textsc{getInstances}(C_i \sqcap C_j) = \emptyset \\ &\quad \wedge \textsc{getSubClasses}(C_i \sqcap C_j) \neq \emptyset \\ \iff &\textsc{testSubClassOf}(C_i, \neg C_j) = \text{incoherent} \end{aligned}$$

By the contrapositive of proposition 8, none of the calls to testSubClassOf return inconsistent, and therefore the maximum value of r which is eventually returned is incoherent. □

Theorem 3. testDisjointClasses *is correct and terminating.*

Proof. Correctness is shown as in the proof of theorem 1. The algorithm terminates because all loops are bounded by n and all calls to testSubClassOf terminate. □

Algorithm 4 tests the disjoint union axiom.

Algorithm 4 test DisjointUnion$(N, C_1, \ldots, C_n)$

Input: N named class
$C_1, \ldots, C_n$ class expressions
$n \geq 2$

```
1: function testDisjointUnion(N, C_1, ..., C_n)
2:     r_1 ← testEquivalentClasses(N, C_1 ⊔ ... ⊔ C_n)
3:     r_2 ← testDisjointClasses(C_1, ..., C_n)
4:     return max(r_1, r_2)
5: end function
```

Theorem 4. testDisjointUnion *is correct and terminating.*

Proof. Correctness is clearly seen from the definition of DisjointUnion, which states separately that $N \equiv C_1 \sqcup \ldots \sqcup C_n$ and DisjointClasses$(C_1, \ldots, C_n)$. Termination is trivial. □

6.2 Assertions

Begin by noting that the addition of an assertion does not affect satisfiability of classes, so O cannot become incoherent. We take this as given, without proof, for all axioms tested in this section.

In this section, when an algorithm accepts n individuals as arguments, we use the shorthand $\mathbf{a} = \{a_1, \ldots, a_n\}$.

Algorithm 5 tests equivalence of individuals. We use the integer variable i to iterate over the indices of individuals given as arguments, and the variable δ to temporarily store a set of individuals.

Algorithm 5 test $a_1 \equiv \ldots \equiv a_n$

Input: $a_1, \ldots, a_n$ individuals
$n \geq 2$

```
 1: function testSameIndividual(a_1, ..., a_n)
 2:     if {a_2, ..., a_n} ⊆ getSameIndividuals(a_1) then
 3:         return entailed
 4:     else
 5:         for i ← 1 to n do
 6:             δ ← getDifferentIndividuals(a_i)
 7:             if {a_1, ..., a_n} ∩ δ ≠ ∅ then
 8:                 return inconsistent
 9:             end if
10:         end for
11:         return absent
12:     end if
13: end function
```

Proposition 12. testSameIndividual *is sound and complete with respect to entailment.*

Proof. Suppose the algorithm returns entailed. This can only happen from line 3, inside the first if-condition, which is reached if and only if

$$\begin{aligned} & \{a_2, \ldots, a_n\} \subseteq \text{getSameIndividuals}(a_1) \\ \iff & (\forall a_j \in \{a_2, \ldots, a_n\}) \;\; a_j \in \text{getSameIndividuals}(a_1) \\ \iff & (\forall a_j \in \{a_2, \ldots, a_n\}) \;\; a_j \equiv a_1 \end{aligned}$$

Equivalence of individuals is transitive, so this is the same as $a_1 \equiv \ldots \equiv a_n$ being entailed. The chain of implication holds in both directions, giving both soundness and completeness. □

Proposition 13. testSameIndividual *is sound with respect to inconsistency.*

Proof. The algorithm can only return inconsistent at line 8, so the first if-condition is false, and for some iteration of the for-loop the inner if-condition is true. From the inner condition, we have

$$\begin{aligned} & (\exists i \leq n) \;\; \mathbf{a} \cap \text{getDifferentIndividuals}(a_i) \neq \emptyset \\ \iff & (\exists a_i, a_j \in \mathbf{a}) \;\; a_j \in \text{getDifferentIndividuals}(a_i) \\ \iff & (\exists a_i, a_j \in \mathbf{a}) \;\; O \vdash a_i \not\equiv a_j \end{aligned}$$

Which contradicts the axiom under test, and therefore

$$\text{test}_O(a_1 \equiv \ldots \equiv a_n) = \text{inconsistent} \quad \square$$

Proposition 14. testSameIndividual *is complete with respect to inconsistency.*

Proof. Suppose the algorithm does not return inconsistent. If it returns entailed, by proposition 12, $\text{test}_O(a_1 \equiv \ldots \equiv a_n) = \text{entailed}$. If not, then for every iteration of the for-loop the inner if-condition must be false. That is,

$$(\forall a_i, a_j \in \mathbf{a}) \;\; O \nvdash a_i \not\equiv a_j$$

It follows that O has an interpretation in which $a_1 \equiv \ldots \equiv a_n$, so $O \cup (a_1 \equiv \ldots \equiv a_n)$ is consistent.

In both cases, $\text{test}_O(a_1 \equiv \ldots \equiv a_n) \neq \text{inconsistent}$, so by contrapositive the proposition holds. □

Theorem 5. testSameIndividual *is correct and terminating.*

Proof. The algorithm is sound and complete with respect to entailment and inconsistency, and by default also absence. All loops are bounded. □

Algorithm 6 tests difference of individuals.

Algorithm 6 test DifferentIndividuals$(a_1, \ldots, a_n)$

Input: $a_1, \ldots, a_n$ individuals
$n \geq 2$

```
 1: function testDifferentIndividuals(a_1, ..., a_n)
 2:     for i ← 1 to n do
 3:         γ ← getSameIndividuals(a_i)
 4:         if ({a_1, ..., a_n} \ {a_i}) ∩ γ ≠ ∅ then
 5:             return inconsistent
 6:         end if
 7:     end for
 8:     for i ← 1 to n do
 9:         δ ← getDifferentIndividuals(a_i)
10:         if ({a_1, ..., a_n} \ {a_i}) ⊈ δ then
11:             return absent
12:         end if
13:     end for
14:     return entailed
15: end function
```

Proposition 15. testDifferentIndividuals *is sound and complete with respect to entailment.*

Proof. The algorithm can only return entailed at line 14. To reach this line, the two if-conditions must be false for every iteration of their enclosing for-loops, else another result would be returned. This means

$$\begin{aligned} & (\forall a_i \in \mathbf{a}) \;\; (\mathbf{a} \setminus \{a_i\}) \cap \text{getSameIndividuals}(a_i) = \emptyset \\ \wedge \; & (\forall a_i \in \mathbf{a}) \;\; (\mathbf{a} \setminus \{a_i\}) \subseteq \text{getDifferentIndividuals}(a_i) \end{aligned}$$

Equivalently, for all distinct arguments $a_i, a_j \in \mathbf{a}$ with $i \neq j$

$$\begin{aligned} & a_j \notin \text{getSameIndividuals}(a_i) \\ & \quad \wedge a_j \in \text{getDifferentIndividuals}(a_i) \\ \iff & O \nvdash a_i \equiv a_j \;\wedge\; O \vdash a_i \not\equiv a_j \end{aligned}$$

This holds if and only if the ontology already entails the axiom. □

Proposition 16. testDifferentIndividuals *is sound with respect to inconsistency.*

Proof. The algorithm can only return inconsistent at line 5. So for some iteration of the first for-loop, its inner if-condition is true. That is,

$$(\exists a_i \in \mathbf{a}) \;\; (\mathbf{a} \setminus \{a_i\}) \cap \text{getSameIndividuals}(a_i) \neq \emptyset$$

So there exist distinct $a_i, a_j \in \mathbf{a}$ such that

$$\begin{aligned} & a_j \in \text{GETSAMEINDIVIDUALS}(a_i) \\ \iff & O \vdash a_i \equiv a_j \end{aligned}$$

This contradicts the axiom under test, so the ontology would become inconsistent if the axiom were added. □

PROPOSITION 17. TESTDIFFERENTINDIVIDUALS *is complete with respect to inconsistency.*

PROOF. Suppose the algorithm does not return inconsistent. Then for every iteration of the first for-loop its inner if-condition must be false. That is,

$$(\forall a_i \in \mathbf{a}) \;\; (\mathbf{a} \setminus \{a_i\}) \cap \text{GETSAMEINDIVIDUALS}(a_i) = \emptyset$$

So for all distinct $a_i, a_j \in \mathbf{a}$ we have $O \nvdash a_i \equiv a_j$. There exists an interpretation of O in which the individuals are all different, which then also models

$$O \cup \text{DifferentIndividuals}(a_1, \ldots, a_n)$$

Therefore ontology with the axiom added is not inconsistent, so by contrapositive the proposition holds. □

THEOREM 6. TESTDIFFERENTINDIVIDUALS *is correct and terminating.*

PROOF. As for theorem 5. □

Algorithm 7 tests the assertion that an individual is an instance of a class expression.

Algorithm 7 test $a : C$

Input: C class expression
a individual

```
1: function TESTCLASSASSERTION(C, a)
2:     if a ∈ GETINSTANCES(C) then
3:         return entailed
4:     else if a ∈ GETINSTANCES(¬C) then
5:         return inconsistent
6:     else
7:         return absent
8:     end if
9: end function
```

PROPOSITION 18. TESTCLASSASSERTION *is sound and complete with respect to entailment.*

PROOF. The algorithm can only return entailed at line 3. This is reached if and only if the first if-condition is true. That is,

$$\begin{aligned} & a \in \text{GETINSTANCES}(C) \\ \iff & O \vdash a : C \\ \iff & \text{test}_O(a : c) = \text{entailed} \end{aligned}$$ □

PROPOSITION 19. TESTCLASSASSERTION *is sound with respect to inconsistency.*

PROOF. Suppose the algorithm returns inconsistent. This can only happen at line 5, so the first if-condition is false and the second is true. That is,

$$\begin{aligned} & a \notin \text{GETINSTANCES}(C) \wedge a \in \text{GETINSTANCES}(\neg C) \\ \iff & O \nvdash a : C \;\wedge\; O \vdash a : \neg C \end{aligned}$$

This contradicts $a : C$, so $O \cup (a : C)$ is inconsistent. □

PROPOSITION 20. TESTCLASSASSERTION *is complete with respect to inconsistency.*

PROOF. Suppose the algorithm does not return inconsistent. If it returns entailed, then by proposition 18 we have that $\text{test}_O(a : C) = \text{entailed}$. If not, then the second if-condition must be false.

$$\begin{aligned} & a \notin \text{GETINSTANCES}(\neg C) \\ \iff & O \nvdash a : \neg C \end{aligned}$$

So O has an interpretation which also models $O \cup (a : C)$ which is therefore consistent. By contrapositive, the proposition holds. □

THEOREM 7. TESTCLASSASSERTION *is correct and terminating.*

PROOF. As for theorem 5. □

6.3 Object property axioms

We present all algorithms in this section without proof of correctness, as the OWL 2 specification [11] describes how the respective object property axioms can be written as class axioms.

Algorithms 8 and 9 test that an object property expression has a given class expression as its domain and range, respectively.

Algorithm 8 test ObjectPropertyDomain(R, C)

Input: R object property expression
C class expression

```
1: function TESTOBJECTPROPERTYDOMAIN(R, C)
2:     return TESTSUBCLASSOF(∃R.⊤, C)
3: end function
```

Algorithm 9 test ObjectPropertyRange(R, C)

Input: R object property expression
C class expression

```
1: function TESTOBJECTPROPERTYRANGE(R, C)
2:     return TESTSUBCLASSOF(⊤, ∀R.C)
3: end function
```

Algorithm 10 tests that an object property expression is functional, and algorithm 11 that it is inverse-functional.

Algorithm 10 test FunctionalObjectProperty(R)

Input: R object property expression

```
1: function FUNCTIONALOBJECTPROPERTY(R)
2:     return
   TESTSUBCLASSOF(⊤, ObjectMaxCardinality(1, R))
3: end function
```

Algorithm 11 test InverseFunctionalObjectProperty(R)

Input: R object property expression

```
1: function INVERSEFUNCTIONALOBJECTPROPERTY(R)
2:     return
   TESTSUBCLASSOF(⊤, ObjectMaxCardinality(1, R⁻))
3: end function
```

Algorithm 12 tests that an object property expression is reflexive, and algorithm 13 that it is irreflexive.

Algorithm 12 test ReflexiveObjectProperty(R)

Input: R object property expression

1: **function** ReflexiveObjectProperty(R)
2: **return** testSubClassOf($\top$, ObjectHasSelf(R))
3: **end function**

Algorithm 13 test IrreflexiveObjectProperty(R)

Input: R object property expression

1: **function** IrreflexiveObjectProperty(R)
2: **return** testSubClassOf(ObjectHasSelf(R), $\bot$)
3: **end function**

7. DEMONSTRATION OF USE CASES

Here we present three examples of axioms being tested to illustrate how the algorithms are used and how they work. We use an ontology O which consists of the following axioms.

$$\text{DisjointClasses}(\texttt{Animal}, \texttt{Plant})$$
$$\texttt{Giraffe} \sqsubseteq \texttt{Mammal}$$
$$\texttt{Mammal} \sqsubseteq \texttt{Animal}$$
$$\texttt{Herbivore} \equiv \forall \texttt{eats}.\, \texttt{Plant}$$
$$\texttt{Susan} : \texttt{Giraffe}$$

Example 1 is straightforward and falls into the use case of testing something we expect to be entailed to assure the quality of the ontology.

Example 1. Test that `Giraffe` is a subclass of `Animal`. In terms of our formal model, this means finding the result of

$$\text{test}_O(\texttt{Giraffe} \sqsubseteq \texttt{Animal})$$

In OWL 2 functional syntax, the axiom is represented as

$$\text{SubClassOf}(\texttt{Giraffe}, \texttt{Animal})$$

So we call algorithm 1.

$$\text{testSubClassOf}(\texttt{Giraffe}, \texttt{Animal})$$

The algorithm first checks if there are any instances of the class expression $\texttt{Giraffe} \sqcap \neg\texttt{Animal}$ at line 2. There are none in this ontology, so it proceeds to check at line if the same class expression has any named subclasses or equivalent classes 4. Again there are none, so it checks if the class expression is satisfiable at line 6. It is not, so the algorithm returns entailed.

Next, examples 2 and 3 demonstrate testing of axioms which are not entailed. This could arise from mistakes in the ontology or in a test, or from temporary tests used for investigation.

Example 2. Test that $\exists\texttt{eats}.\,\texttt{Animal} \sqsubseteq \texttt{Herbivore}$. As in example 1, we call algorithm 1.

$$\text{testSubClassOf}(\exists\texttt{eats}.\,\texttt{Animal}, \texttt{Herbivore})$$

This checks if $\exists\texttt{eats}.\,\texttt{Animal} \sqcap \neg\texttt{Herbivore}$ has instances at line 2, which it does not, then named subclasses at line 4, which it does not. Then it checks if it is satisfiable at line 6, which it is because the ontology does not entail that it is empty, so the algorithm returns absent. This means that the axiom is not entailed but it would not cause inconsistency or incoherence if added to the ontology.

Note that it is not possible to test this axiom with any of the tools identified in section 3 because the left hand side of the axiom, the subclass, is not a named class.

Example 3. Test that `Susan` : `Plant`. Now we call algorithm 7.

$$\text{testClassAssertion}(\texttt{Plant}, \texttt{Susan})$$

This first checks if `Susan` is a known instance of `Plant` at line 2, which it is not. Then it checks if `Susan` is an instance of $\neg\texttt{Plant}$ at line 4, which it is because `Giraffe` is disjoint with `Plant`, so the algorithm returns inconsistent.

8. DISCUSSION

The algorithms we have presented fully cover class axioms and partially cover assertions and object property axioms. They lay the groundwork for testing algorithms to be devised for the remaining axioms, and for their correctness to be proven.

These new testing algorithms address the shortcomings of the existing tools TDDonto, Tawny-OWL, and Scone. Specifically, they offer greater coverage of OWL 2 axioms, and return more detailed test results. The have also been rigorously proven to be correct. They will facilitate the development of new tools, or the extension of existing ones, which in turn can aid the adoption of test-driven development within ontology engineering, and then hopefully the adoption of ontologies within business and industry.

TDDonto already implements a testing harness with a broad collection of testing algorithms. It could easily be modified to make use of these new algorithms. It could then support the testing of those complex axioms which it does not currently support, and report detailed information on the consequences of adding a new axiom. This would make it an invaluable component of the ontology development workflow within Protégé.

Even though Tawny-OWL tests are predicates which must only return boolean results, they could still make use of slightly modified versions of the new algorithms so that they support the same complex axioms, only return boolean results, and include detailed results in a full test report.

Scone could likewise be extended to support testing class axioms which contain complex expressions, and object and data properties.

There is scope for ontology testing algorithms to be improved further. Firstly, certain constrained cases might be tested more efficiently. For example, testing $N \sqsubseteq C$ where N is a named class, it suffices to determine entailment by checking if $N \in \text{getSubClasses}(C)$, which may be faster. Secondly, it may be possible to test axioms more efficiently when the ontology is known to be in a restricted profile of OWL 2. Lastly, the algorithms could be extended to generate justifications of inconsistency or incoherence without the need to reclassify the ontology.

When implementing these algorithms, there are certain considerations. Most importantly, the consistency and coherence preconditions should be checked once before evaluating a suite of tests, and missing entities should be checked

before each test. Next, it should be made possible to test isSatisfiable(C) directly so as to allow authors to ensure satisfiability of arbitrary class expressions. Lastly, since some reasoners support checking isEntailed(A), it may make sense to use it where possible. This merits investigation and benchmarking.

9. CONCLUSION

We have presented a novel model of ontology testing. We have presented a comprehensive collection of algorithms to be used for testing axioms within ontologies, and proven them to be correct. Finally we have demonstrated the applicability of these algorithms with pertinent use cases.

The clearest scope of future work is to implement these algorithms in a new or existing tool. Work also remains in benchmarking any implementations against other tools, to verify that they are comparably fast, and in improving their efficiency in known cases such as simple axioms or restricted profiles.

10. REFERENCES

[1] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[3] Cucumber. https://cucumber.io/. Accessed 1 November 2016.

[4] Gene Ontology Consortium. Gene ontology consortium: going forward. *Nucleic Acids Research*, 43(D1):D1049–D1056, 2015.

[5] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang. HermiT: An OWL 2 reasoner. *Journal of Automated Reasoning*, 53(3):245–269, 2014.

[6] R. Iqbal, M. A. A. Murad, A. Mustapha, and N. M. Sharef. An analysis of ontology engineering methodologies: A literature review. *Research Journal of Applied Sciences, Engineering and Technology*, 6(16):2993–3000, 2013.

[7] M. Kaczmarek. Ontologies in the realm of enterprise modeling—a reality check. In *Formal Ontologies Meet Industry*, volume 225 of *Lecture Notes in Business Information Processing*, pages 39–50. Springer International Publishing, 2015.

[8] C. M. Keet. Detecting and revising flaws in OWL object property expressions. In *Knowledge Engineering and Knowledge Management*, volume 7603 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2012.

[9] C. M. Keet and A. Ławrynowicz. Test-driven development of ontologies. In *The Semantic Web: Latest Advances and New Domains*, volume 9678 of *Lecture Notes in Computer Science*, pages 642–657. Springer, 2016.

[10] A. Ławrynowicz and C. M. Keet. The TDDonto tool for test-driven development of DL knowledge bases. In *29th International Workshop on Description Logics*, number 1577 in CEUR Workshop Proceedings, 2016.

[11] B. Motik, P. F. Patel-Schneider, and B. Parsia. OWL 2 web ontology language structural specification and functional-style syntax (second edition). W3C recommendation, W3C, 2012.

[12] OWL API. http://owlcs.github.io/owlapi/. Accessed 1 November 2016.

[13] OWL-BGP. https://www.uni-ulm.de/en/in/ki/software/owl-bgp.html. Accessed 9 May 2016.

[14] Y. Rafique and V. B. Mišić. The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering*, 39(6):835–856, 2013.

[15] D. Ramsden. What did linked data ever do for us anyway? http://www.bbc.co.uk/academy/technology/software-engineering/semantic-web/article/art20130720153136618, 2013. Accessed 4 May 2016.

[16] Scone project. https://bitbucket.org/malefort/scone. Accessed 9 May 2016.

[17] E. Simperl, M. Mochol, T. Bürger, and I. O. Popov. Achieving maturity: The state of practice in ontology engineering in 2009. In *On the Move to Meaningful Internet Systems: OTM 2009*, volume 5871 of *Lecture Notes in Computer Science*, pages 983–991. Springer, 2009.

[18] D. Vrandečić and A. Gangemi. Unit tests for ontologies. In *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, volume 4278 of *Lecture Notes in Computer Science*, pages 1012–1020. Springer, 2006.

[19] J. D. Warrender and P. Lord. How, what and why to test an ontology. In *Bio-Ontologies 2015*, 2015.

APPENDIX

A. UNTESTED AXIOMS

Here we provide a complete list of OWL 2 axioms for which this paper does not describe a testing algorithm.

- Assertions
  - ObjectPropertyAssertion
  - NegativeObjectPropertyAssertion
  - DataPropertyAssertion
  - NegativeDataPropertyAssertion
- Object property axioms
  - SubObjectPropertyOf
  - EquivalentObjectProperties
  - DisjointObjectProperties
  - InverseObjectProperties
  - SymmetricObjectProperty
  - AsymmetricObjectProperty
  - TransitiveObjectProperty
- Data property axioms
  - SubDataPropertyOf
  - EquivalentDataProperties
  - DisjointDataProperties
  - DataPropertyDomain
  - DataPropertyRange
  - FunctionalDataProperty
- Annotation axioms
  - AnnotationAssertion
  - SubAnnotationPropertyOf
  - AnnotationPropertyDomain
  - AnnotationPropertyRange
- Other
  - Declaration
  - DatatypeDefinition
  - HasKey