TDDonto2: A Test-Driven Development Plugin for arbitrary TBox and ABox axioms

Kieren Davies[1], C. Maria Keet[1], and Agnieszka Ławrynowicz[2]

[1] Department of Computer Science, University of Cape Town, South Africa
kdavies@cs.uct.ac.za, mkeet@cs.uct.ac.za
[2] Institute of Computing Science, Poznan University of Technology, Poland
agnieszka.lawrynowicz@cs.put.poznan.pl

Abstract. Ontology authoring is a complex task where modellers rely heavily on the automated reasoner for verification of changes, using effectively a time-consuming test-last approach. Test-first with Test-Driven Development aims to speed up such processes, but tools to date covered only a subset of possible OWL 2 DL axioms and provide limited feedback. We have addressed these issues with a model for TDD testing to give more feedback to the modeller and seven new, generic, TDD algorithms that also cover OWL 2 DL class expressions on the left-hand side of inclusions and ABox assertions by availing of several reasoner methods. The model and algorithms have been implemented as a Protégé plugin, TDDonto2.

1 Introduction

With most automated reasoners for OWL having become stable and reliable over the years, ontology engineers are exploring their creative uses to assist the ontology authoring process of ontology development. For instance, the possible world explorer examining negations [4], the entailment differences of an ontology edit [3, 8], and proposing the feasible object properties [7]. This is in a considerable part motivated by the time-consuming trial-and-error approach in the authoring process where many modellers invoke the reasoner even after each single edit [11], noting also that aforementioned methods still require classification for each assessment step. Such practices are unsustainable when the ontology becomes large or complex and classifying the ontology prohibitively long. Analysing such modeller behaviour, this actually amounts to a *test-last* mode, alike unit testing in software development. In that regard, ontology engineering methodologies lag behind software engineering methodologies in terms of both maturity and adoption [5]. In particular, there is only one tentative methodology that explicitly incorporates automated testing as a *test-first* approach (that reduces the number of times a reasoner has to be invoked) [6], which is a staple of software engineering as test-driven development (TDD) [1]. There are a few tools for TDD unit testing ontologies in this manner [6, 12, 10], i.e., (in short) checking whether an axiom is entailed before adding it. They all share two notable shortcomings, however: certain axioms are not supported as TDD unit tests even though they

are permitted in OWL 2, such as $\forall R.C \sqsubseteq D$, and test results are mostly just "pass" or "fail" with no further information about the nature of failure. Moreover, no rigorous theoretical analysis of the techniques used for such test-first ontology testing has been carried out. However, for modellers to be able to fully rely on reasoner-driven TDD in the ontology authoring process—as they do with test-last ontology authoring—such rigour is an imperative.

In this demo-paper, we present TDDOnto2, which fills this gap in rigour and coverage. It relies on a succinct logic-based model of TDD unit testing as a prerequisite and generalised versions of the algorithms of [6] to cover also *any* OWL 2 class expression in the axiom under test for not only the TBox, as in [6], but also ABox assertions. The model details and proofs of correctness of the algorithms are described in [2]. These algorithms do not require reclassification of an ontology in any test after a first single classification before executing one or more TDD unit test, and are such that the algorithms are compliant with any OWL 2 compliant reasoner. This is feasible through 'bypassing' the ontology editor functionality and availing directly of a set of methods available from the OWL reasoners in a carefully orchestrated way.

We have implemented both the model for testing and the novel algorithms by extending TDDonto [6] as a proof-of-concept to ascertain their correct functioning practically. It uses the OWL API [9] and a subset of its functions, including isSatisfiable(C), getSubClasses(C), getInstances(C), and getTypes(a), for the 'convenience method' isEntailed is not mandatory for reasoners to implement, and most do not. This open source Protégé 5 plugin, TDDonto2, is accessible at `https://github.com/kierendavies/tddonto2`, which also has a screencast of the working code. A screenshot is included in Fig. 1.

The remainder of this demo paper describes several scenarios where TDD aspects are useful (Section 2), and then introduces TDDonto2 and illustrates several of its algorithms through brief examples (Section 3). We close with conclusions and what an attendee may expect from the demo (Section 4).

2 Scenarios for testing during ontology development

Ontologies, like computer programs, can become complex so that it is difficult for a human author to predict the consequences of changes. Automated tests are therefore useful to detect unintended consequences. For instance, suppose an author creates the following classes and subsumptions: `Giraffe` $\sqsubseteq$ `Herbivore` $\sqsubseteq$ `Mammal` $\sqsubseteq$ `Animal`, but then realises that not all herbivores are mammals, so shortens the hierarchy to `Herbivore` $\sqsubseteq$ `Animal`, thereby losing the `Giraffe` $\sqsubseteq$ `Mammal` derivation. An application that uses this ontology to retrieve mammals would then erroneously exclude giraffes. This issue can be caught by a simple automated test to check whether `Giraffe` $\sqsubseteq$ `Mammal` is still entailed. It may seem like this problem can be solved just adding those axioms directly to the ontology. However, adding such axioms introduces a lot of redundancy, making modification of the ontology more difficult. Adding only a test instead ensures correctness without bloating the ontology.

Tests may also be used 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, saving the time it would take to browse the inferred class hierarchy. A similar approach can be employed when developing a new ontology: create a temporary test to determine whether the axiom i) is already entailed, ii) would result in a contradiction or unsatisfiable class if it were to be added to the ontology, or iii) can be added safely. The standard approach of adding an axiom and then observing the consequences involves reclassification, which is typically very slow, and which a TDD unit test can avoid.

Overall, there are thus two broad use cases: 1) Declare many tests alongside an ontology and evaluate them in order to demonstrate quality or detect regressions; 2) Evaluate temporary tests in order to explore an ontology or predict the consequences of adding a new axiom. Such scenarios are made possible with test-driven development with the TDDonto2 tool.

3 Illustration of TDDonto2's algorithms

The simple workflow for an actual test in TDDonto2 is to type an axiom in the test text box; e.g., eats some Animal SubClassOf: (Carnivore or Omnivore), and either "Evaluate" it immediately (as with giraffe SubClassOf: mammal in Fig. 1) or "Add" it to the test suite (middle of the screen), then either select a subset of the tests (Shift-/Ctrl-click), or all tests, and test them by pressing the "Evaluate selected" or "Evaluate all" button, respectively. The "Result" can be one of the following: the knowledge is already in the ontology (*entailed*), adding the axiom will make the ontology *inconsistent*, adding the axiom will make at least one class unsatisfiable (*incoherent*), the axiom is *absent* and will not lead to a contradiction if added, and *failed precondition* for if the ontology is already inconsistent or incoherent. Based on the results, one either can "Remove" the axiom under test or "Add selected to ontology".

In the remainder of this section we demonstrate examples of axioms being tested so as to illustrate how the algorithms are used and how they work (see also Fig. 1). Take a simple ontology O that consists of the following axioms:

$\texttt{Giraffe} \sqsubseteq \texttt{Mammal}$	$\texttt{Carnivore} \sqsubseteq \texttt{Animal}$
$\texttt{Mammal} \sqsubseteq \texttt{Animal}$	$\texttt{Carnivore} \sqcap \texttt{Herbivore} \sqsubseteq \bot$
$\texttt{Animal} \sqcap \texttt{Plant} \sqsubseteq \bot$	$\texttt{Susan} : \texttt{Giraffe}$
$\texttt{Herbivore} \equiv \texttt{Animal} \sqcap \forall \texttt{eats}.\texttt{Plant}$	$\texttt{Max} : \texttt{owl:Thing}$

Example 1 is straightforward and falls into the use case of testing something a modeller expects to be entailed to ensure the quality of the ontology.

Example 1. Test that $\texttt{Giraffe}$ is a subclass of $\texttt{Animal}$, hence, finding the result of $\text{test}_O(\texttt{Giraffe} \sqsubseteq \texttt{Animal})$. It first checks if there are any instances of the class expression $\texttt{Giraffe} \sqcap \neg\texttt{Animal}$. There are none in this ontology, so it proceeds to check if the same class expression has any named subclasses or equivalent classes. Again there are none, so it checks if the class expression is satisfiable. It is not, so the algorithm returns entailed. $\diamondsuit$

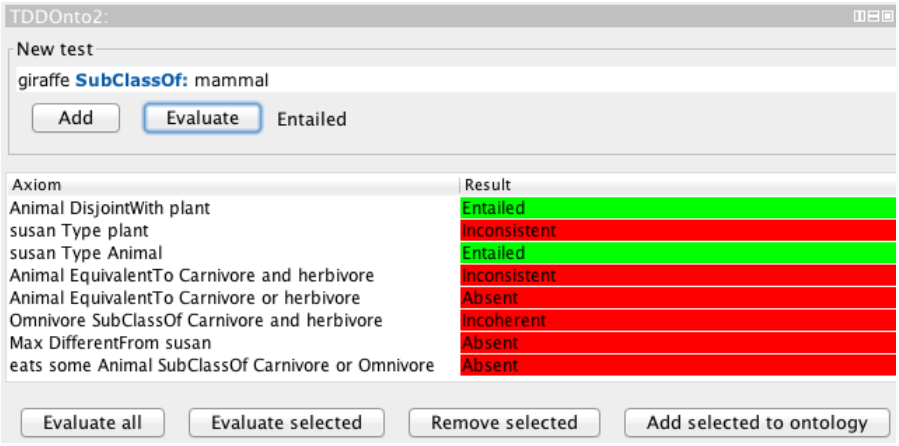

Fig. 1. Screenshot of TDDonto2 after having run several TDD unit tests on a sample ontology. Top: entering a test; middle: tests and their results; bottom: managing tests.

Examples 2, 3, and 4 described below demonstrate testing of more interesting axioms that are not possible to test with any of the extant TDD tools, for i) the left-hand side of the inclusion is not a named class (Ex. 2), ii) have a test with individuals (Ex. 3 and 4), and iii) the axioms are not entailed for different reasons.

Example 2. Test that $\exists \texttt{eats}.\texttt{Animal} \sqsubseteq \texttt{Carnivore}$. First, the algorithm checks if $\exists \texttt{eats}.\texttt{Animal} \sqcap \neg\texttt{Carnivore}$ has instances (if so, then the ontology with this axiom would be inconsistent), which it does not, and then if it has named subclasses, which it does not (so, the ontology with this axiom would not cause the ontology to become incoherent). Then it checks if it is satisfiable, which it is because the ontology does not entail that it is empty, so the algorithm returns absent. Thus, the axiom is not entailed and it would not cause inconsistency or incoherence if added to the ontology. ♢

Example 3. Test whether $\texttt{Susan} : \texttt{Plant}$. TDDonto2 first checks if `Susan` is a known instance of `Plant`, which it is not. Then it checks if `Susan` is an instance of $\neg\texttt{Plant}$, which it is because `Giraffe` is disjoint with `Plant` because $\texttt{Giraffe} \sqsubseteq \texttt{Animal}$ and $\texttt{Animal} \sqcap \texttt{Plant} \sqsubseteq \bot$, so the algorithm returns inconsistent. ♢

Example 4. Test whether `Susan` and `Max` are different individuals. It first retrieves all the individuals that are the same as `Susan`; this set is empty, so adding the different individuals axiom will not result in an inconsistent ontology. Then it retrieves all the individuals different from `Susan`; `Max` is not in that set, so the algorithm will return 'absent', hence, the axiom can be added without causing the ontology to be come inconsistent and without introducing redundancy. ♢

More examples illustrating the tool and a screencast are available from `https://github.com/kierendavies/tddonto2`.

4 Conclusions and Demo

The algorithms implemented in TDDonto2 fully cover class axioms and partially cover assertions and object property axioms. They significantly broaden the coverage compared to the existing tools [6, 12, 10] and return more detailed test results. TDDonto2 easily could be extended or integrated with generating justifications of inconsistency or incoherence without the need to reclassify the ontology, and return more user-friendly explanations alike in [3].

In the demo, we will illustrate all possible permutations of the testing model's possible return values, as well as its coverage of types of axioms, and that it indeed does reduce the number of calls to the reasoner, hence, reduces ontology authoring time. Attendees can bring their own ontology and try it out, and we also will have several ontologies an attendee can test with.

Acknowledgments This work has been partially supported by the National Science Centre, Poland, within grant 2014/13/D/ST6/02076. A. Ławrynowicz acknowledges support from grant 09/91/DSPB/0627.

References

1. Beck, K.: Test-Driven Development: by example. Addison-Wesley, Boston, MA (2004)
2. Davies, K.: Towards test-driven development of ontologies: An analysis of testing algorithms. Project report, University of Cape Town (2016), `https://people.cs.uct.ac.za/~dvskie001/doc/TDD_Ontologies_Analysis_of_Testing_Algorithms.pdf`
3. Denaux, R., Thakker, D., Dimitrova, V., Cohn, A.G.: Interactive semantic feedback for intuitive ontology authoring. In: Proc. of FOIS'12. pp. 160–173. IOS Press (2012)
4. Ferré, S., Rudolph, S.: Advocatus diaboli exploratory enrichment of ontologies with negative constraints. In: Proc. of EKAW'12. LNAI, vol. 7603, pp. 42–56. Springer (2012), 8-12 Oct 2012, Galway, Ireland
5. Iqbal, R., Murad, M.A.A., Mustapha, A., Sharef, N.M.: An analysis of ontology engineering methodologies: A literature review. Research Journal of Applied Sciences, Engineering and Technology 6(16), 2993–3000 (2013)
6. Keet, C.M., Ławrynowicz, A.: Test-driven development of ontologies. In: Proc. of ESWC'16. LNCS, vol. 9678, pp. 642–657. Springer (2016)
7. Keet, C.M., Khan, M.T., Ghidini, C.: Ontology authoring with FORZA. In: Proc. of CIKM'13. pp. 569–578. ACM proceedings (2013)
8. Matentzoglu, N., Vigo, M., Jay, C., Stevens, R.: Making entailment set changes explicit improves the understanding of consequences of ontology authoring actions. In: Proc. EKAW'16. LNAI, vol. 10024, pp. 432–446. Springer (2016)
9. OWL API. `http://owlcs.github.io/owlapi/`, accessed: 1-11-2016
10. Scone project. `https://bitbucket.org/malefort/scone`, accessed: 9-5-2016
11. Vigo, M., Bail, S., Jay, C., Stevens, R.D.: Overcoming the pitfalls of ontology authoring: strategies and implications for tool design. International Journal of Human-Computer Studies 72(12), 835–845 (2014)
12. Warrender, J.D., Lord, P.: How, what and why to test an ontology. In: Bio-Ontologies 2015 (2015)