Unit tests for ontologies

Denny Vrandečić[1], Aldo Gangemi[2]

[1]denny@aifb.uni-karlsruhe.de
Institute AIFB, University of Karlsruhe, Germany
[2]aldo.gangemi@istc.cnr.it
LOA Laboratory of Applied Ontology, Rome, Italy

Abstract. In software engineering, the notion of unit testing was successfully introduced and applied. Unit tests are easy manageable tests for small parts of a program – single units. They proved especially useful to capture unwanted changes and side effects during the maintenance of a program, and they grow with the evolution of the program.
Ontologies behave quite differently than program units. As there is no information hiding in ontology engineering, and thus no black box components, at first the idea of unit testing for ontologies seems not applicable. In this paper we motivate the need for unit testing, describe the adaptation to the unit testing approach, and give use cases and examples.

1 Introduction

In software engineering, the idea of unit testing [1] was introduced to counter the complexities of modern software engineering efforts. Unit tests are meant to facilitate the development of program modules or units, and to ensure the interplay of such units in the combined system. It results in more loosely coupled code, that is easier to refactor and simpler to integrate, and that has a formalized documentation (although not necessarily complete). Unit tests can be added incrementally during the maintenance of a piece of software, in order to not accidentally stumble upon and old bug and hunt it down repeatedly.

Unit tests are not complete test suites: there are several types of errors that unit tests will not catch, including errors that result from the integration of the units to the complete system, performance problems, and, naturally, errors that were not expected when writing the unit tests.

Unit tests in software engineering became popularized with the object oriented language Smalltalk, and still to this today remain focused on languages with strong possibilities to create smaller units of code. They are based on several decomposition techniques, most important of all information hiding.

Ontologies are different. As of now, no form of information hiding or interfaces are available – and it remains an open research issue in the field of ontology modularization how this will be taken care of.

In this paper we will take a look at the benefits of unit testing applied to ontologies, i.e. their possibilities to facilitate regression tests, to provide a test framework that can grow incrementally during the maintenance and evolution

phase of the ontology, and that is reasonably simple to use. In order for the unit testing for ontologies to be useful, they need to be reasonably easy to use and mantain. This will depend heavily on the given implementation (which is underway). The task of this paper is to investigate different ideas that are inspired by the idea of unit testing, and to work out the intuitions of how these ideas could be used in the context of the ontology lifecycle. Especially in the enterprise application of ontologies, some easy to use form of ontology evaluation will be required in order to let ontology based technologies become more widespread. We will show a number of ideas and examples of how this goal can be achieved.

The paper will first show a typical use case, as encountered in a project setting in Section 2. We will then discuss five different approaches, that all are inspired by the unit testing frameworks in software engineering: first we look at the idea of design by contract, i.e. of stating what statements should and should not derive from an ontology being developed or maintained, either as explicit ontology statements or as competency questions using a query language (Sections 3 and 4). Then we investigate the relationship of heavy- to lightweight ontologies, and how they can interplay with regards to ontology evaluation in Section 5. Autoepistemic operators lend themselves also to be used in the testing of ontologies, especially with regards to their (relative) completeness, since they are a great way to formalize the introspection of ontologies (Section 6). We also regard a common error in ontology modelling with description logics based language, and try to turn this error into our favour in Section 7, before we discuss related work and give an outlook on possible further work and open issues.

For this work, the term ontologies refers to web ontologies as defined by the OWL DL standard [15]. This means that the ontologies are a variant based on the description logics $\mathcal{SHOIN}(\mathbf{D})$, and, especially, that ontologies mean to encompass both the so called TBox, where the vocabulary of the ontology is defined (which some call the whole ontology), and the ABox, where facts using the vocabulary defined are stated (which some call the knowledge base).

2 Motivation

In the SEKT project[1], one of the case studies aims at providing an intelligent FAQ system to help newly appointed judges in Spain [2]. The system depends on an ontology for finding the best answers and to find references to existing cases in order to provide the judge with further background information. The applied ontology is built and maintained by legal experts with almost no experience in formal knowledge representation [4].

As the ontology evolved and got refined (and thus changed), the legal experts noticed that some of their changes had undesired side effects. To give a simplified example, consider the class hierarchy depicted in Figure 1. Let's take for granted that this ontology has been used for a while already, before someone notices that not every academic needs necessarily be a member of an university. So *Academic*

[1] `http://www.sekt-project.com`

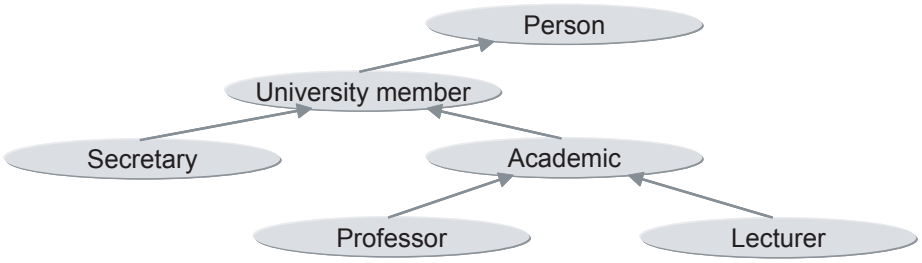

Fig. 1. Example class hierarchy

becomes a direct subclass of *Person*, instead of *University member*. But due to this change, also *Professor* is no subclass of *University member* any more (a change that maybe was hidden from the ontology engineer, as the ontology development environment may not have displayed the subclasses of *Academic*).

The resulting ontology remains perfectly satisfiable. But a tool, that, for example, creates a web page for all members of the university may now skip the professors, since they are not classified as university members any more – an error that would only become apparent in the use of the tool much later and will be potentially hard to track down to that particular ontology change operation.

Unit testing for ontologies can discover such problems, and a few other ones as well, as we will see in the following sections.

3 Affirming derived knowledge

We create two test ontologies T^+ (called the positive test ontology) and T^- (the negative test ontology), and define that an ontology O, in order to fulfil the constraints imposed by the test ontologies, needs to fulfil the following conditions: each axiom $A_1^+ ... A_n^+ \in T^+$ must be derivable from O, i.e.

$$O \models A_i^+ \forall A_i^+ \in T^+$$

and each axiom $A_1^- ... A_n^- \in T^-$ must not be derivable from O, i.e.

$$O \not\models A_i^- \forall A_i^- \in T^-$$

Note that T^+ trivially fulfils the first condition if O is not satisfiable, whereas an empty ontology trivially fulfils the second condition. So it is not hard to come up with ontologies that fulfil the conditions, which shows that unit tests are not meant to be complete formalizations of the requirements of an ontology, but rather helpful indicators towards possible errors or omissions in the tested ontologies. Note also that T^- could be unsatisfiable, i.e. there are two sets of axioms (both subsets of T^- that contradict each other. This still makes sense, as it means that O must not make a decision about the truth of either of these

sets (thus formalizing the requirement that O must be agnostic towards certain statements).

To come back to our previous example in Section 2 a simple test ontology T^+ that consists of the single axiom $Professor \sqsubseteq Universitymember$ would have been sufficient to discover the problem described. So after the discovered error, this statement is added to the test ontology, and now this same error will be detected by running the unit tests.

The test ontologies are meant to be created and grown during the maintenance of the ontology. Every time an error is encountered in the usage of the ontology, the error is formalized and added to the appropriate ontology (like in the example above). Experienced ontology engineers may add appropriate axioms in order to anticipate and counter possible errors in maintenance.

In software engineering it is often the case, that the initial development of a program is done by a higher skilled, better trained, and more consistent team, whereas the maintenance is then performed by a less expensive group, with less experienced members, that change more frequently. So in software engineering, the more experienced developers often anticipate frequent errors that can happen during maintenance, and create unit tests accordingly in order to put appropriate constraints on the future evolution of the software. We expect a similar development in ontology engineering and maintenance, as soon as ontologies become more common components of information systems. The framework proposed in this paper offers the same possibilities to an ontology engineer.

Why should an ontology engineer not just add the axioms from T^+ to O, and $\neg A_i^-$ for each A_i^- in T^-? There are several reasons: 1) not every axiom A_i^- can be negated. For example, the simple statement $R(a, b)$ stating a relation R between the individuals a and b can not be negated in OWL DL. 2) adding such axioms increases redundancy in the ontology, and thus makes it harder to edit. 3) the axioms may potentially increase reasoning complexity, or else use language constructs that are not meant to be used within the ontology, for whatever reason. 4) as stated above, the axioms in T^- may be contradictory. 5) Finally, due to the open world assumption, $O \not\models A_i^- \forall A_i^- \in T^-$ is not the same as $O \models \neg A_i^- \forall A_i^- \in T^-$, so that the negative test ontology can actually not be simulated with the means of OWL DL.

4 Formalized competency questions

Competency questions, as defined by some methodologies for ontology engineering (like OTK [17] or Methontology [6]), describe what kind of knowledge the resulting ontology is supposed to answer. These questions can always be formalized in a query language (or else the ontology will actually not be able to answer the given competency question, and thus will not meet the given requirements). Formalizing the queries, instead of writing them down in natural language, and formalizing the expected answers as well, allows for a system that automatically checks if the ontology meets the requirements stated with the competency questions.

We consider this approach especially useful not for the maintenance of the system, but rather for its initial build, in order to define the extent of the ontology. Note that competency questions usually are just exemplary questions – answering all competency questions does not mean that the ontology is complete. Also note that sometimes, although the question is formalizable, the answer does not necessarily need to be known at the time of writing the question. This is especially true for dynamic ontologies, i.e. ontologies that reflect properties of the world that keep changing often (like the song the user of the system is listening to at query time). In that case we can define some checks if the answer is sensible or even possible (like that the answer indeed needs to be a song). Often these further checks will not go beyond the abilities defined by the other approaches to unit testing in this paper.

5 Expressive consistency checks

Ontologies in information systems often need to fulfil the requirement of allowing reasoners to quickly answer queries with regards to the ontology. Light weight ontologies usually fulfil this task best. Also, many of the more complex constructors of OWL DL often do not add further information, but rather are used to restrict possible models. This is useful in many applications, like ontology mapping and alignment, or information integration from different sources.

For example, a minimal cardinality constraint will, due to the open world assumption, hardly ever lead to any inferred statements in an OWL DL ontology (this can only become the case if range of the minimal cardinality restricted relation is a class consisting of nominals). Nevertheless the statement can be useful as an indicator for tools that want to offer a user interface to the ontology, or for mapping algorithms that can take this information into account.

Further expressive constraints on the ontology, like disjointness of classes, can be used to check the ontology for consistency at the beginning of the usage, but after this has been checked, a light weight version of the ontology, that potentially enables reasoners to derive answers with a better response time, could be used instead.

Also, for these pre-use consistency checks, more expressive logical formalisms could be used, like reasoning over the ontology metamodel [13, 3], using SWRL [11], or using the transformation of the ontology to a logic programming language like datalog [8] and then add further integrity constraints to that resulting program (that may easily go far beyond the expressivity available in the original ontology language: with a logic programming language it would be easy to state that, by company policy, a supervisor is required to have a higher income than the persons reporting to her – which is impossible in OWL DL).

Formally, we introduce a test ontology T^C for an ontology O, that includes the high axiomatization of the terms used in O, and check for the satisfiability of the merged ontology $O \cup T^C$. In the case of using the logic programming translation, we merge (concatenate) the translation of the ontology to datalog

($LP(O)$) with the test program T^C_{LP}, and test the resulting program for violation of the integrity constraints.

Let us consider an example ontology O_x:
$tradesStocks(Jim, MegaCorp)$
$CEO(MegaCorp, Jack)$
$bestBuddy(Jim, Jack)$
$bestBuddy \sqsubseteq hasNoSecrets$

This example ontology is equivalent to the translated logic program $LP(O_x)$:
$tradesStocks(Jim, MegaCorp).$
$CEO(MegaCorp, Jack).$
$bestBuddy(Jim, Jack).$
$hasNoSecrets(X, Y) :- bestBuddy(X, Y).$

The test program T^C_{LP} (that checks for insider trading) may consist of the following line, a single integrity constraint:
$: - hasNoSecrets(X, Y), tradesStocks(Y, C), CEO(X, C).$

Evaluating the program should now raise a violated integrity constraint. We could also name the integrity constraints (by putting $insiderTrading$ into its head and then explicitly evaluate $insiderTrading$ to see if it evaluates to true or false. We also could use the head $insiderTrading(Y)$ and then query for the same head, to get a list of people who actually do the insider trading (and thus uncover problems in the program much faster).

6 Use of autoepistemic operators

In [7] an extension of OWL DL with autoepistemic operators is described. Especially the **K**-operator can be useful to check an ontology not only with regards to its consistent usage, but also with regards to some explicitly defined understanding of completeness. In a geographic ontology, we may define that every country has a capital, $Country \sqsubseteq \exists capital.City$. But stating the existence of a country without stating its capital will not lead to an error in the ontology, because the reasoner (correctly) assumes, that the knowledge is not complete. Using the **K**- and **A**−operators instead, we would define that $\mathbf{K}Country \sqsubseteq \exists \mathbf{A}capital.\mathbf{A}City$, i.e. for every known country the capital must also be known (i.e. either stated explicitly or inferrable) in the ontology, or else the ontology will be not satisfiable (the example follows ex. 3.3 in [5]). Thus we are able to state what *should* be known, and a satisfiability check will check if, indeed this knowledge is present.

On the Semantic Web, such a formalism will prove of great value, as it allows to simply discard data that does not adhere to a certain understanding of completeness. For example, a crawler may gather event data on the Semantic Web. But instead of simply collecting all instances of event, it may decide to only accept events that have a start and an end date, a location, a contact email, and a classification with regards to a certain term hierarchy. Although this will decrease the recall of the crawler, the data will be of a higher quality, i.e. of a bigger value, as it can be sorted, displayed, and actually used by calendars, map tools, and email clients in order to support the user.

The formalisation and semantics of autoepistemic operators for the usage in web ontologies is described in [7], and thus will not be repeated here.

7 Domain and ranges as constraints

Users often complain that their intuitive usage of relation domain and ranges contradict their actual semantics. They expect domain and ranges to be used like constraints, i.e. if they say that *brideOf* is a relation with the domain *Woman* and one applies it to *John*, who is a *Man*, instead of getting an inconsistency *John* will be classified as a *Woman* by the reasoner (for the sake of the example, we take it for granted that *Man* and *Woman* are not disjoint). In the following we will try to use this error to the benefit of the user.

As the domain and range declarations in the ontology will usually render these checks trivially true, we need to remove them first from the ontology. Therefore we take an ontology, and delete all domain and range declarations that are to be understood as constraints (in the example in the previous paragraph, we would remove the declaration of *brideOf*'s domain as *Woman*). Now we check for all instantiations of the removed domain or range declaration's relation if the subject (in case of a removed domain declaration) or the object (in case of a removed range declaration) indeed gets classified with the supposed class (in the example, we ask if *John* is indeed a *Woman*).

Note that these removals may have further implications on the ontology's inferred statements, depending on the further axioms of the ontology, and its planned usage. Experiments need to be performed to be able to judge the described approach with regards to its impact on the inferred knowledge in real world scenarios. This approach actually will not necessarily highlight errors, but only indicate possible places for errors. It will probably make more sense to introduce a new relation that let us define constraints for relations, and then to check these explicitly. We expect to learn from the planned experiments how to exactly bring this approach to use.

8 Related work

A Protégé Plug-In implementing an OWL Unit Test framework[2] exists, that allows to perform what we have described with T^+ testing for affirming derived knowledge in Section 3.

In [16] the theory and practice of ontology evolution is discussed. Ontology change operations and ontology evolution strategies are introduced. Based on this, [9] extends this work for OWL DL ontologies, and investigates the evolution of ontologies with regards to consistency, implemented in the so called evOWLution framework. As the theoretical work allows generic and user defined consistency checks, the ideas presented here could be regarded as a number of

[2] `http://www.co-ode.org/downloads/owlunittest/`

ways to formalize further aspects of the ontology, and enable more expressive consistency checks beyond simple logical satisfiability.

Some parts of the presented ideas – especially the test ontologies in Section 3 and the consistency check against heavy weight descriptions in Section 5 – may often lead to unsatisfiable ontologies when the unit testing uncover problems. In this case, research done in debugging [14] and revising [12], especially evolving ontologies [10], will provide the tools and techniques to actually resolve the problems.

9 Outlook

The approaches described in this paper address problems in ontology engineering and maintenance that have been discovered during the work with ontologies within the SEKT case studies. As they often reminded us of problems that occurred in software engineering, a solution that was successfully introduced to software engineering was examined – unit testing. Although the notion of unit testing needed to be changed, it inspired a slew of possible approaches, that have been described in this paper. Also, examples for these approaches have been given to illustrate how they can be used within the lifecycle of an ontology.

As of now, we are working on an implementation of the presented ideas in order to experimentally verify their usefulness. Although we have shown that several problems we have encountered can be solved with the presented approaches, it is unclear if the idea behind them is simple enough to be understood by non-experts in ontology engineering. Also it needs to be investigated, how often certain classes of problems appear in real world ontologies, and which of the ideas presented here are most effective to counter these problems.

Like in software engineering, we do not expect unit tests to cover the whole field of ontology evaluation. But we expect it to become (and remain) an important building block within an encompassing framework, that will cover regression tests and (relative) completeness, and help to indicate further errors in the initial development, and especially further maintenance of an ontology.

We expect modularization of ontologies and networked ontology to become more important in the next generation of web ontology based technologies. Unit testing provides a framework to formalize requirements about ontologies. We expect the approaches described in this paper, and maybe further similar approaches, to become much more investigated and discussed in the close future.

Acknowledgements

Research reported in this paper was supported by the IST Programme of the European Community under the SEKT project, Semantically Enabled Knowledge Technologies (IST-1-506826-IP, `http://www.sekt-project.com`). We want to thank our colleagues for fruitful discussions, especially Stephan Grimm, Boris Motik, Peter Haase, Sudhir Agarwal, Jos Lehmann, and Malvina Nissim.

This publication reflects the authors' view.

References

1. K. Beck. Simple Smalltalk testing: With patterns. `http://www.xprogramming.com/testfram.htm`.
2. V. Benjamins, P. Casanovas, J. Contreras, J. L. Cobo, and L. Lemus. Iuriservice: An intelligent frequently asked questions system to assist newly appointed judges. In V. Benjamins, P. Casanovas, A. Gangemi, and B. Selic, editors, *Law and the Semantic Web*, LNCS, Berlin Heidelberg, 2005. Springer.
3. S. Brockmans and P. Haase. A Metamodel and UML Profile for Rule-extended OWL DL Ontologies –A Complete Reference. Technical report, Universität Karlsruhe, March 2006. `http://www.aifb.uni-karlsruhe.de/WBS/sbr/publications/owl-metamodeling.pdf`.
4. P. Casanovas, N. Casellas, M. Poblet, J. Vallbé, Y. Sure, and D. Vrandečić. Iuriservice ii ontology development. In P. Casanovas, editor, *Workshop on Artificial Intelligence and Law at the XXIII. World Conference of Philosophy of Law and Social Philosophy*, May 2005.
5. F. M. Donini, D. Nardi, and R. Rosati. Description logics of minimal knowledge and negation as failure. *ACM Transactions on Computational Logic*, 3(2):177–225, 2002.
6. M. Fernández-López, A. Gómez-Pérez, J. P. Sierra, and A. P. Sierra. Building a chemical ontology using Methontology and the Ontology Design Environment. *IEEE Intelligent Systems*, 14(1), 1999.
7. S. Grimm and B. Motik. Closed world reasoning in the semantic web through epistemic operators. In B. C. Grau, I. Horrocks, B. Parsia, and P. Patel-Schneider, editors, *OWL: Experiences and Directions*, Galway, Ireland, Nov 2005.
8. B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20-24 May 2003*, pages 48–57. ACM, 2003.
9. P. Haase and L. Stojanovic. Consistent evolution of OWL ontologies. In A. Gómez-Pérez and J. Euzenat, editors, *Proc. of the 2nd European Semantic Web Conf. ESWC*, volume 3532, pages 182–197, Heraklion, Crete, Greece, 2005. Springer.
10. P. Haase, F. van Harmelen, Z. Huang, H. Stuckenschmidt, and Y. Sure. A framework for handling inconsistency in changing ontologies. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *Proceedings of the ISWC2005*, volume 3729 of *LNCS*, pages 353–367. Springer, NOV 2005.
11. I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: a semantic web rule language combining OWL and RuleML, 2003.
12. S. J. Lam, D. Sleeman, and W. Vasconcelos. Retax++: a tool for browsing and revising ontologies. In *Proc. ISWC 2005 Demo Session*, Galway, Ireland, 2005.
13. B. Motik. On the properties of metamodeling in OWL. In *Proc. of the 4th International Semantic Web Conference, Galway, Ireland, November 2005*, NOV 2005.
14. B. Parsia, E. Sirin, and A. Kalyanpur. Debugging OWL ontologies. In *Proc. of the 14th World Wide Web Conference (WWW2005)*, Chiba, Japan, May 2005.
15. M. K. Smith, C. Welty, and D. McGuinness. OWL Web Ontology Language Guide, 2004. W3C Rec. 10 February 2004, avail. at `http://www.w3.org/TR/owl-guide/`.
16. L. Stojanovic. *Methods and Tools for Ontology Evolution*. PhD thesis, University of Karlsruhe, 2004.
17. Y. Sure and R. Studer. On-To-Knowledge methodology. In J. Davies, D. Fensel, and F. van Harmelen, editors, *On-To-Knowledge: Semantic Web enabled Knowledge Management*, chapter 3, pages 33–46. J. Wiley and Sons, 2002.