The Semantic Web takes Wing: Programming Ontologies with Tawny-OWL

Phillip Lord

School of Computing Science, Newcastle University

Abstract. The Tawny-OWL library provides a fully-programmatic environment for ontology building; it enables the use of a rich set of tools for ontology development by recasting development as a form of programming. It is built in Clojure – a modern Lisp dialect, and is backed by the OWL API. Used simply, it has a similar syntax to OWL Manchester syntax, but it provides arbitrary extensibility and abstraction. It builds on existing facilities for Clojure, which provides a rich and modern programming tool chain, for versioning, distributed development, build, testing and continuous integration. In this paper, we describe the library, this environment and the its potential implications for the ontology development process.

1 Introduction

Ontology building remains a difficult and demanding task. Partly this is intrinsic, but also stems from the tooling. For example, while ontology editors like Protégé [1] do allow manual ontology development, they are not ideal for automation or template-driven development; as a result, languages such as OPPL[2] have been developed which allow a slightly higher-level of abstraction over the base OWL axiomatisation. However, they involve a move away from OWL syntax, which in turn requires integration into which ever environment the developers are using. There has also been significant interest in collaborative development of ontologies, either using collaborative development tools such as Web-Protege[3], or through copy-modify-merge versioning[4].

In this work, we[1] take an alternative approach. Instead of developing tools for ontology development, many of which are similar or follow on from software development tools, we attempt to recast ontology development as a software engineering problem, and then simply reuse the standard tools that exist for software engineering. We have achieved this by developing a library, named *Tawny OWL*, that at its simplest operates as a domain specific language for OWL, while still retaining the full capabilities of a modern programming language with all this entails. We demonstrate the application of this library to a standard exemplar - namely the Pizza Ontology[5], as well as several other scenarios. Finally, we consider the implications of this approach for enabling collaborative and more agile forms of ontology development.

[1] Plurals are used throughout, and do not indicate multiple authorship.

The Tawny-OWL library is being developed on GitHub (`https://github.com/phillord/tawny-owl`); it currently consists of around 3000 lines of code, and supports OWL2 object and annotation properties.

2 Requirements

Interaction between OWL and a programming API is not a new idea. For example, OWL2Perl[6] allows generation of Perl classes from an OWL Ontology, FuXi does similar in Python `http://code.google.com/p/fuxi/`, while the OWL API allows OWL ontology development in Java[7]. The OWL API, however, is rather unwieldy for direct ontology development; for example, it has a complex type hierarchy, indirect instantiation of objects through factories, and a set of change objects following a command design pattern; while these support one of its original intended use case – building a GUI – they make direct ontology development cumbersome. One response to this is Brain[8,9], which is a much lighter-weight facade over the OWL API also implemented in Java. Brain is, effectively, typeless as expressions are generated using Strings; the API distinguishes between OWL class creation (`addClass`) and retrieval (`getClass`), throwing exceptions to indicate an illegal state. While Brain is useful, it is not clear how an ontology should be structured in Java's object paradigm, and it suffers the major drawback of Java – an elongated compile-test-debug cycle, something likely to be problematic for interactive development as the ontology increases in size.

For programmatic ontology development, we wanted an interactive and dynamic environment rather like the R environment for statistics, where the ontology could be explored, extended and reworked on-the-fly. For this reason we choose to build in Clojure; a modern Lisp derivative with many attractive features: persistent data structures; specialised mechanisms for state. It suffers somewhat from being built on the Java Virtual Machine (JVM) – this gives it a rather slow start-up time. However, in this case, it was a key reason for its use. Interoperability with the JVM is integrated deeply into Clojure which makes building on top of the OWL API both possible and convenient; this interoperability means any feature of the OWL API can be used within tawny, without moving from the Lisp syntax; so, while tawny does not currently wrap for isntance either datatype properties, nor ontology explanation code, both are still easily accessible. Like all lisps, Clojure has three other advantages: first, it is untyped which, in common with Brain, in this context, we consider to be an advantage[2]; second, it is highly dynamic – almost any aspect of the language can be redefined at any time – and it has a full featured read-eval-print-loop (REPL); finally, it has very little syntax, so libraries can manipulate the look of the language very easily. Consider, for example, a simple class definition as shown in Listing1, taken from a pizza ontology available at `https://github.com/phillord/tawny-pizza`. The syntax has been designed after Manchester syntax[10].

[2] We do not argue that type systems are bad; just that they are less appropriate in this environment

```
(defclass Pizza
   :label "Pizza"
   :comment
"An␣oven-baked␣flat␣bread␣with␣toppings,␣originating␣from␣Italy."
)
```

Listing 1. A basic class definition

A more complex definition shows the generation of restrictions and anonymous classes.

```
(defclass CheesyPizza
  :equivalent
  (owland Pizza
          (owlsome hasTopping CheeseTopping)))
```

Listing 2. A Cheesy Pizza

These definitions bind a new symbol (`Pizza` and `CheesyPizza`) to a OWL-API Java object. These symbols resolve as a normal Var does in Clojure. Strictly, this binding is not necessary (and can be avoided if the user wishes), however this provides the same semantics as Brain's `addClass` and `getClass` – classes, properties, etc must be created before use; a valuable feature protecting against typing errors[11].

2.1 Lisp Terminology

Here we give a brief introduction to Clojure and its terminology. Like all lisps, it has a regular syntax consisting of parenthesis delimited `(lists)`, defining an *expression*. The first element is usually a *function*, giving lisps a prefix notation. Elements can be literals, such as strings *e.g.* `"Pizza"`, *symbols e.g.* `defclass` or *keywords e.g.* `:equivalent`. Symbols resolve to their values, keywords resolve to themselves, and literals are, well, literal. Unlike many languages, these constructs are directly manipulable in the language itself which combined with *macros* enable extension of the language.

3 A Rich Development Environment

There are a dizzying array of ontology development tools available[12]. Probably the most popular is Protégé; while it provides a very rich environment for viewing and interacting with an ontology, it lacks many features that are present in most IDEs. For instance, it lacks support for version control or adding to ChangeLogs; it is not possible to edit documentation along side the ontology; nor to edit code in other languages when, for instance, driving a build process, or using an ontology in an application.

We have previously attempted to work around this problem by providing support for Manchester syntax – *OMN* – within Emacs through *omn-mode*[13]; while this provides a richer general-purpose environment, the ontology environment is comparatively poor. In particular, only syntactic completion is available

and there is no support for documentation look-up beyond file navigation. Finally, we used Protégé (and the OWL API) to check syntax, which required a complete re-parse of the file, and with relatively poor feedback from Protégé when errors occurred[3].

With tawny, using a general purpose programming language, a richer development environment comes nearly for free. In this paper, we describe the use within Emacs; however, support for Clojure is also available within Eclipse, IntelliJ, Netbeans and other environments[14]. Compared with direct editing of OMN files, tawny provides immediate advantages. The use of paren delimiters makes indentation straight-forward, well-defined, and well-supported; advanced tools like paredit ensures that expressions are always balanced. Clojure provides a REPL, and interaction within this allows more semantic completion of symbols even when they are not syntactically present in the buffer[4], which is common when using levels of abstraction (Section 4) or external OWL files (Section 8). Syntax checking is easy, and can be performed on buffer, marked region or specific expression. New entities can be added or removed from the ontology on-the-fly without reloading the entire ontology, enabling progressive development. We have also provided support for documentation look-up of OWL entities, hooked into Clojure's native documentation facility so should function within all development environments. We do not currently provide a rich environment for browsing ontologies, except at the code level; however, Protégé works well here, reloading OWL files when they are changed underneath it. Similarly, omn-mode can be used to view individual generated OMN files.

4 Supporting Higher Levels of Abstraction

Most ontologies include a certain amount of "boilerplate" code, where many classes follow a similar pattern. Tools such as OPPL were built specifically to address this issue; with tawny, the use of a full programming language, makes the use of levels of abstraction above that in OWL straight-forward. We have used this in many areas of tawny; at its simplest, by providing convenience macros. For example, it is common-place to define many subclasses for a single superclass; using OMN each subclass must describe its superclass. Within tawny, a dynamically-scoped block can be used as shown in Listing 3. As shown here, disjoint axioms can also be added[15]; and, not used here, covering axioms[16]. The equivalent OMN generated by these expressions is also shown in Listing 4.

```
(as-disjoint-subclasses
 PizzaBase

 (defclass ThinAndCrispyBase
   :annotation (label "BaseFinaEQuebradica" "pt"))
```

[3] This is not a criticism of the Protégé interface; it was not designed to operate on hand-edited files

[4] We follow Emacs terminology here – a *buffer* is a file being edited

```
(defclass DeepPanBase
  :annotation (label  "BaseEspessa" "pt")))
```

Listing 3. Subclass Specification

```
Class: piz:ThinAndCrispyBase
    Annotations:
        rdfs:label "BaseFinaEQuebradica"@pt
    SubClassOf:
        piz:PizzaBase
    DisjointWith:
        piz:DeepPanBase

Class: piz:DeepPanBase
    Annotations:
        rdfs:label "BaseEspessa"@pt,
    SubClassOf:
        piz:PizzaBase
    DisjointWith:
        piz:ThinAndCrispyBase
```

Listing 4. Subclasses in OMN

It is also possible to add suffixes or prefixes to all classes created within a lexical scope. For example, we can create classes ending in `Topping` as shown in Listing 5. While similar functionality could be provided with a GUI, this has the significant advantage that the developers intent remains present in the source; so subsequent addition of new toppings are more likely to preserve the naming scheme.

```
(with-suffix Topping
   (defclass GoatsCheese)
   (defclass Gorgonzola)
   (defclass Mozzarella)
   (defclass Parmesan))
```

Listing 5. Adding Suffixes

Tawny also includes initial support for ontology design patterns; in particular, we have added explicit support for the value partition[17]. This generates classes, disjoints and properties necessary to fulfil a pattern, but is represented in tawny succinctly (Listing 6)

```
(p/value-partition
 Spiciness
 [Mild
  Medium
  Hot])
```

Listing 6. A Value Partition

While some abstractions are generally useful, an important advantage of a full-programmatic language for OWL is that abstractions can be added to

any ontology including those which are likely to be useful only within a single ontology. These can defined as functions or macros in the same file as their use. For example, within the pizza ontology, Listing 7 generates two pizzas – in each case the pizza class comes first, followed by its constituent parts; a closure axiom is added to each pizza. As well, as being somewhat more concise than the equivalent OMN, this approach also has the significant advantage that it is possible to change the axiomatisation for all the named pizzas by altering a single function; this is likely to increase the consistency and maintainability of ontologies.

```
(generate-named-pizza
 [MargheritaPizza MozzarellaTopping TomatoTopping]

 [CajunPizza MozzarellaTopping OnionTopping PeperonataTopping
  PrawnsTopping TobascoPepperSauce TomatoTopping]
```

Listing 7. Generating Named Pizzas

5 Separating Concerns for Different Developer Groups

One common requirement in ontology development is a separation of concerns; different contributors to the ontology may need different editing environments, as for instance with RightField or Populous[18]. tawny enables this approach also; here, we describe how this enables internationalisation. Originally, the pizza ontology had identifiers in English and Portuguese but, ironically, not Italian. While it would be possible to have a translator operate directly on a tawny source file, this is not ideal as they would need to need to embed their translations within OWL entity definitions as shown in Listing 3; which is likely to be particularly troublesome if machine assisted translation is required due to the non-standard format. We have, therefore added support with the *polyglot* library. Labels are stored in a Java properties file (Listing 8) and are loaded using a single Lisp form (Listing 9). tawny will generate a skeleton resources file, with no translations, on demand, and reports missing labels to the REPL on loading.

```
AnchoviesTopping=Acciughe Ingredienti
ArtichokeTopping=Carciofi Ingredienti
AsparagusTopping=Asparagi Ingredienti
```

Listing 8. Italian Resources

```
(tawny.polyglot/polyglot-load-label
  "pizza/pizzalabel_it.properties" "it")
```

Listing 9. Loading Multi-Lingual Labels

Currently, only loading labels is supported in this way, but extending this to comments or other forms of annotation is possible. While, in this case, we are loading extra-logical aspects of the ontology from file, it would also be possible to load logical axioms; for instance, named pizzas (Section 4) could be loaded from text file, spreadsheet or database.

6 Collaborative and Distributed Development

Collaborative development is not a new problem; many software engineering projects involve many developers, geographically separated, in different time zones, with teams changing over time. Tools for enabling this form of collaboration are well developed and well supported. Some of these tools are also available for ontology development; for instance, Web-Protégé enables online collaborative editing. However, use of this tool requires installation of a bespoke Tomcat based server, nor does it yet support offline, concurrent modification[3].

Alternatively, the ContentCVS system does support offline concurrent modification. It uses the notion of structural equivalence for comparison and resolution of conflicts[4]; the authors argue that an ontology is a set of axioms. However, as the named suggests, their versioning system mirrors the capabilities of CVS – a client-server based system, which is now considered archaic.

For tawny, the notion of structural equivalence is not appropriate. With tawny, the axioms are generated, rather than being source; the source cannot, therefore, be abstracted to a set of axioms. A single change in the source file, might result in 1000s of changes in the axiomatisation. Additionally, programmer intent is often represented through non-axiomitised sections of the code – whitespace, indentation and even comments which may drive a "literate" development approach. Moreover, a definition of a difference based purely on axiomatisation cannot account for these differences; the use of a line-oriented syntactic diff will.

We argue here that by provision of an attractive and well-supported syntax, we do not need to provide specific collaborative tooling. Tawny itself has been built using distributed versioning systems (first Mercurial and then git). These are already advanced systems supporting multiple workflows including tiered development with authorisation, branching, cherry-picking and so on. While ontology-specific tooling may have some advantages, it is unlikely to replicate the functionality offered by these systems, aside from issues of developer familiarity.

Later, we also describe support for automated testing, which can also ease the difficulty of collaborative working (Section 9).

7 Enabling Modularity

Tawny provides explicit support for name spacing and does this by building on Clojure's namespace support. It is possible to build a set of ontologies spread across a number of different files. Normally, each file contains a single namespace; tawny mirrors this, with each namespace containing a single ontology, with a defined IRI.

OWL itself does not provide a distribution mechanism for ontologies; the IRI of an ontology does not need to resolve, but in practice, is often a distribution mechanism. By default Protégé will check for resolution if other mechanisms fail; OBO ontologies, for example, are all delivered from their IRI.

In contrast, tawny builds on the Clojure environment; most projects are built using the Leiningen tool which, in turn, uses the repository and dependency management from Maven. When building the Pizza ontology in tawny, the build tool

will fetch tawny itself, the OWL API and HermiT, as well as their dependencies. Ontological dependencies can be fetched likewise. Maven builds come with a defined semantics for versioning, including release and snapshot differentiation. A key advantage of this system is that multiple versions of a single ontology can be supported, with different dependency graphs.

8 Coping With Semantics Free Identifiers

Importing one ontology from another is straight-forward in tawny. However, not all ontologies are developed using tawny; we need to be able interact with external ontologies only accessible through an OWL file. Tawny provides facilities for this use-case: the library reads the OWL file, creates symbols for all entities[5], then associates the relevant Java object with this symbol. This approach is reasonably scalable; tawny can import the Gene Ontology within a minute on a desktop machine. Clojure is a highly-dynamic language and allows this form of programmatic creation of variables as a first-class part of the language; so an ontology read in this way functions in every sense like a tawny native ontology. Ontology classes can be queried for their documentation, auto-completion works and so forth.

However, there is a significant problem with this import mechanism. Tawny must create a symbol for each OWL entity in the source ontology. By default, tawny uses the IRI fragment for this purpose; while Clojure symbols have a restricted character set which is not the same as that of the IRI fragment, in practice this works well. However, this is unusable for ontologies built according to the OBO ontology standard, which uses semantics-free, numeric identifiers such as `OBI_0000107`. While these are valid Clojure symbols, it is unreadable for a developer. This issue also causes significant difficulties for ontology development in any syntax; OMN is relatively human-readable but ceases to be so when all identifiers become numeric. We have previously suggested a number of solutions to this problem, either through the use of comments or specialised denormalisations[19], or through the addition of an `Alias` directive providing a mapping between numeric and readable identifier[20]. However, all of these require changes to the specification and tooling updates, potentially in several syntaxes.

For tawny, we have worked around this problem by enabling an arbitrary mapping between the OWL entity and symbol name [21]. For OBO ontologies, a syntactic transformation of the `rdfs:label` works well. Thus, `OBI_0000107` can be referred to as `provides_service_consumer_with`, while `BFO_0000051` becomes the rather more prosaic `has_part`.

While this solves the usability problem, it causes another issue for ontology evolution; the label is open to change, independently of any changes in semantics; unfortunately, any dependent ontology built with tawny will break, as the relevant symbol will no longer exist. This problem does not exist for GUI editors such as Protégé because, ironically, they are not WYSIWYG – the ontology

[5] It is possible to choose a subset

stores an IRI, while the user sees the label; changes to labels percolate when reloading the dependent ontology. Tawny provides a solution to this; it is possible to *memorise* mappings between symbols and IRIs at one point in time[22]. If the dependency changes its label, while keeping the same IRI, tawny will recognise this situation, and generate a *deprecated* symbol; dependent ontologies will still work, but will signal warnings stating that a label has changed and suggesting appropriate updates. Currently these must be performed manually, although this could be automated.

9 Enabling Unit Testing and Continuous Integration

Unit testing is a key addition to the software development process which has enabled more agile development. Adapting this process for ontology development has previously been suggested[23], and implemented as a plugin to Protégé [24]. To add this capability to tawny, we have integrated reasoning; at the time of writing, only ELK[25] is available as a Maven resource in the Maven Central repository, therefore we have developed a secondary maven build for HermiT which allows use of this reasoner also[26][6], so both these reasoners are available for use; others can be added trivially as they are *mavenised*. A number of test frameworks exist in Clojure; here we use `clojure.test`. As shown in Listing 10, we check that various inferences have occurred (or not as appropriate), using the `isuperclass?` predicate. We have also added support for "probe" classes. In our second test, we use the `with-probe-entities` macro; this adds a subclass of `VegetarianPizza` and `CajunPizza` – as the latter contains meat, this should result in an incoherent ontology if both definitions are correct; probe entities are automatically removed by the macro, returning the ontology to its previous state, ensuring independence of tests.

```
(deftest CheesyShort
  (is (r/isuperclass? p/FourCheesePizza p/CheesyPizza))
  (is (r/isuperclass? p/MargheritaPizza p/CheesyPizza))
  (is
   (not (r/isuperclass? p/MargheritaPizza p/FourCheesePizza))))

(deftest VegetarianPizza
  (is
   (r/isuperclass? p/MargheritaPizza p/VegetarianPizza))

  (is
   (not
    (o/with-probe-entities
      [c (o/owlclass "probe"
                     :subclass p/VegetarianPizza p/CajunPizza)]
      (r/coherent?)))))
```

Listing 10. Unit Testing a Pizza Ontology

[6] Available at http://homepages.cs.ncl.ac.uk/phillip.lord/maven/, or on Github

In addition to coherence/consistency checking, unit tests are also useful to check the OWL profile in use; particularly useful when using higher levels of abstraction (see Section 4) which hides the exact axiom in use.

The use of Unit testing in this way has implications beyond simple ontology development; it also allows a richer form of continuous integration where dependent ontologies can be developed by independent developers, but continuously checked for breaking changes. The tawny pizza ontology, for example, is currently being tested using Travis[7]. Unlike, other ontology CI systems[27], this requires no installation and integrates directly with the DVCS in use. It is also useful for integration with software that operates on the ontology; for example, both our version of Hermit, the OWL API and tawny-owl are built and tested using this tool.

10 Discussion

In this paper, we have described tawny, a library which enables the user to develop ontologies, using the tooling and environments that have long been available to programmers. Although they both involve producing artifacts with strong computational properties ontology development and software engineering have long been disjoint. This has significant negative impacts; there are far more programmers than knowledge engineers, and as a result the tooling that they use is far better developed. Tawny seeks to address this imbalance, not by providing richer tools for ontology development, but by recasting ontology development as a form of programming.

By allowing knowledge engineers to use any level of abstraction that they choose, tawny can also improve current knowledge engineering process significantly. It can help to remove duplication, for example, in class names. It can clearly delineate disjoint classes protecting against future additions; this helps to address a common ontological error[28]. It is also possible to model directly using common ontology design patterns generating many axioms in a succinct syntax. Bespoke templates can be built for a specific ontology; this mirrors functionality of tools like OPPL[2], but uses the power of a full programming language and environment. Trivially, for example, tawny can log its activity and comes with debugger support.

Of course, direct use of a programmatic library like tawny is not suitable for all users; however, even for these users a library like tawny could be useful. It is straight-forward to integrate ontologies developed directly with tawny as a DSL with knowledge stored in other formalisms or locations. In this paper, we described loading multi-lingual labels from properties files, isolating the translator from the ontology, and interacting with OWL files generated by another tool. It would also be possible to load axioms from a database or spreadsheet, using existing JVM libraries.

While with tawny, we have provided a programmatic alternative to many facilities that exist in other tools, we also seek to provide tooling for a more ag-

[7] `http://travis-ci.org`

ile and reactive form of ontology development. Current waterfall methodologies, exemplified by BFO-style realism, lack agility, failing to meet the requirement for regular releases to address short-comings, as has been seen with both BFO 1.1[29] and BFO 2.0[30]. Likewise, the OBO foundry places great emphasis on a review process which is, unfortunately, backlogged[31] – in short, as with waterfall software methodologies, the centralised aspects of this development model appear to scale poorly.

Tawny uses many ready-made and well tested software engineering facilities: amenability to modern DVCS, a versioning and release semantics, a test framework and continuous integration. The provision of a test environment is particularly important; while ontology developers may benefit from testing their own ontologies, the ability to contribute tests to their ontological dependencies is even more valuable. They can provide upstream developers precise and executable descriptions of the facilities which they depend on; giving upstream developers more confidence that their changes will not have unexpected consequences. When this does happen, downstream developers can track against older versions of their dependencies, obviating the need for co-ordination of updates; when they do decide to update, the re-factoring necessary to cope with external changes will be supported by their own test sets; finally, continuous integration will provide early warning if a developer's own changes impact others. In short, tawny provides the tools for a more pragmatic and agile form of ontology development which is more suited to fulfilling the changing and varied requirements found in the real world[32].

References

1. Various: The protégé; ontology editor and knowledge acquisition system. `http://protege.stanford.edu/`
2. Egana Aranguren, M., Stevens, R., Antezana, E.: Transforming the axiomisation of ontologies: The ontology pre-processor language. Nature Precedings (Dec 2009)
3. Tudorache, T., Nyulas, C., Noy, N.F., Musen, M.A.: WebProtg: a collaborative ontology editor and knowledge acquisition tool for the web. Semantic Web
4. Jiminez Ruiz, E., Grau, B.C., Horrocks, I., Berlanga, R.: Supporting concurrent ontology development: Framework, algorithms and tool. Data & Knowledge Engineering **70**(1) (Jan 2011) 146–164
5. Stevens, R.: Why the pizza ontology tutorial? `http://robertdavidstevens.wordpress.com/2010/01/22/why-the-pizza-ontology-tutorial/` (2010)
6. Kawas, E., Wilkinson, M.D.: Owl2perl: creating perl modules from owl class definitions. Bioinformatics **26**(18) (Sep 2010) 2357–2358
7. Bechhofer, S., Volz, R., Lord, P.: Cooking the semantic web with the OWL API. In: Internaional Semantic Web Conference. (2003) 659 – 675
8. Croset, S., Overington, J., Rebholz-Schuhman, D.: Brain: Biomedical knowledge manipulation. Bioinformatics (2013) Submitted.
9. loopasam: Brain. `https://github.com/loopasam/Brain`
10. Horridge, M., Patel-Schneider, P.: Owl 2 web ontology language manchester syntax. `http://www.w3.org/TR/owl2-manchester-syntax/` (2012)

11. Lord, P.: Owl concepts as lisp atoms. http://www.russet.org.uk/blog/2254 (2012)
12. Bergman, M.: The sweet compendium of ontology building tools. http://www.mkbergman.com/862/the-sweet-compendium-of-ontology-building-tools/ (2010)
13. Lord, P.: Ontology building with emacs. http://www.russet.org.uk/blog/2161 (2012)
14. Various: Getting started - clojure documentation - clojure development. http://dev.clojure.org/display/doc/Getting+Started
15. Lord, P.: Disjoints in clojure-owl. http://www.russet.org.uk/blog/2275 (2012)
16. Stevens, R.: Closing down the open world: Covering axioms and closure axioms. http://ontogenesis.knowledgeblog.org/1001 (2011)
17. Rector, A.: Representing specified values in owl: "value partitions" and "value sets". W3C Working Group Note (2005)
18. Jupp, S., Horridge, M., Iannone, L., Klein, J., Owen, S., Schanstra, J., Wolstencroft, K., Stevens, R.: Populous: a tool for building owl ontologies from templates. BMC Bioinformatics **13**(Suppl 1) (2011) S5
19. Lord, P.: Obo format and manchester syntax. http://www.russet.org.uk/blog/1470 (2009)
20. Lord, P.: Semantics-free ontologies. http://www.russet.org.uk/blog/2040 (2012)
21. Lord, P.: Clojure owl 0.2. http://www.russet.org.uk/blog/2303 (2012)
22. Lord, P.: Remembering the world as it used to be. http://www.russet.org.uk/blog/2316 (2013)
23. Vrandečić, D., Gangemi, A.: Unit tests for ontologies. In: In OTM Workshops (2
24. Drummond, N.: Co-ode & downloads & the owl unit test framework. http://www.co-ode.org/downloads/owlunittest/ [Online. last-accessed: 2013-01-28 15:22:03]
25. Kazakov, Y., Krtzsch, M., Simancik, F.: Elk reasoner: Architecture and evaluation. In: Proceedings of the 1st International Workshop on OWL Reasoner Evaluation (ORE-2012). (2012)
26. Various: Hermit reasoner: Home. http://hermit-reasoner.com/
27. Mungall, C., Dietze, H., Carbon, S., Ireland, A., Bauer, S., Lewis, S.: Continuous integration of open biological ontology libraries. http://bio-ontologies.knowledgeblog.org/405 (2012)
28. Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., Wroe, C.: OWL pizzas: Practical experience of teaching OWL-DL: common errors & common patterns. Engineering Knowledge in the Age of the Semantic Web (2004) 6381
29. Various: New version of bfo 1.1 available. https://groups.google.com/d/topic/bfo-discuss/HQSnudUUM4E/discussion
30. Various: Proposal for an official bfo 1.2 release. https://groups.google.com/d/topic/bfo-discuss/iKBlfDPv5GM/discussion
31. OBO Foundry Outreach Working Group: New obo foundry tracker for feedback, requests, and other issues. http://sourceforge.net/mailarchive/message.php?msg_id=30391720
32. Lord, P., Stevens, R.: Adding a little reality to building ontologies for biology. PLoS One (2010)