OWL/ZIP: Distributing Large and Modular Ontologies

Nicolas Matentzoglu and Bijan Parsia

The University of Manchester
Oxford Road, Manchester, M13 9PL, UK
{bparsia,matentzn}@cs.manchester.ac.uk

Abstract. Ontologies published on the web can be quite large, from a couple of megabytes to more than a gigabyte. Deploying, importing and using such ontologies can be a problem, both in terms of bandwidth and load time over the web, and in terms of physically storing them. Some ontologies in BioPortal for example are shipped in their compressed form (via their web services), which allows filesize reductions to up to 2% of the original. Moreover, many ontologies have been published in their modular form through the use of owl:imports. Some of the imports are dereferenceable on the web, but in many cases, the imports closure is shipped with the actual (root) ontology, which can lead to confusions on how to interpret the directory structure. In this paper, we are proposing a set of simple conventions to distribute ontologies (monolithic or modular) in a canonical fashion to enable tools to make use of pre-compiled modular structures and reduce IO communication to a minimum using compression.

Keywords: OWL, Ontologies, ZIP, Compression

1 Introduction

Many important ontologies today, such as the Foundational Model of Anatomy (FMA) [2] and the Cell Cycle Ontology (CCO) [1] have grown to filesizes of hundreds of megabytes in their serialised form. Using, importing or just regularly downloading a fresh copy over the web can lead to often severe bottlenecks in load time. Some ontologies available on BioPortal [6], such as the FMA, use compressed archives for distribution, which allows filesize reductions to up to 2% of the original filesize. Tools that automatically retrieve and work with these compressed ontologies will need custom functionality to deal with the archive (i.e. unpack it, identify the correct file to load, load it). In many cases, it would be desirable to simply deploy the ontology in their compressed form, and have standard tool support (such as integration with the OWL API, Protege or Jena) to deal with the correct compression and decompression. Ontologies that are published in their modular form (i.e., a set of ontologies pointing at one another with owl:imports), either logical decompositions or results of a modular engineering process, have often been published as archives (see for example some

examples in BioPortal). These archives however can be ambiguous as to which file corresponds to the actual ontology (i.e., the entry point). In this paper, we are briefly discussing decompositions as a use case for OWL/ZIP, describe the format in more detail, and then presenting a survey of the effects of using compression to deploy ontologies over a large corpus.

2 Dealing with decompositions

A decomposition $\mathcal{D}$ of an ontology $\mathcal{O}$ (a set of axioms) is a partitioning of $\mathcal{O}$ into a set of subsets $\mathcal{S} \subseteq \mathcal{O}$ where it holds that the union of all $\mathcal{S} \subseteq \mathcal{O}$ equals $\mathcal{O}$. There are two fundamentally different ways to decompose an ontology:

1. As part of the engineering process, ontologies are designed to be modular to enable re-use across multiple specialised domains or
2. A decomposition is extracted from the ontology post-engineering to enable services to interact with logically coherent subsets of the ontology rather than having to deal with the whole. In the context of this work this distinction only serves to establish two very different interest groups for OWL/ZIP.

Analysing MOWLCorp, a crawl-based corpus of around 21K ontologies [**?**][1], we can find at least 462 syntactically unique ontologies with more than 5 imports, and 16 with more than 15. All these are examples of the first kind of decomposition, where engineers have decided to deploy the ontology in somehow topically related subsets that might be individually re-used. While imports intended for re-use should be dereferenceable on the web, the reality is that many ontology providers prefer to deploy their modular structure with their ontologies, which creates a need to deal with these kinds of distributions in an unambiguous way. The need for dealing with the second type of decomposition first emerged from rather recent advances in modular reasoning (e.g., maintaining a ontology in decomposed form removes some overhead from a modular reasoning process). A decomposition in this sense is more strictly defined: every subset in the above definition corresponds to a logical (in practice locality-based) module. Instead of duplicating axioms across multiple modules, the modular structure can be encoded in a much terser representation such as the Atomic Decomposition [8], which will guarantee that every axiom is serialised only once, and the modular structure is encoded through a dependency graph between axioms and sets of axioms, called atoms. This dependency graph can be straight forwardly represented as a pattern of imports, where every (directed) edge representing a dependency corresponds to a owl:imports statement. While we can observe import structures with tens of elements in hand-built cases, automated logical decompositions are typically far more complicated. In the worst case, we have around as many ontologies in the imports structure as we have axioms in the original ontology. This also clearly establishes a need to deploy these sorts of decompositions in a compressed and "single file" form.

[1] `http://mowlrepo.cs.manchester.ac.uk/datasets/mowlcorp/`

3 OWL/ZIP Convention

In order to accommodate both for large monolithic ontologies and their decomposed counterparts we mainly need to agree on:

1. a type of compression in order to minimize the effort to develop suitable tool support
2. an entry point in the archive
3. a standard way to determine whether to prefer local imports, if existent, over global ones

Our suggestions is as follows. ZIP is a widely used type of compression supported by most tools that support compressed files in general. Even if the compression rates might not be optimal, size reductions are possible to between 2 and 20% of the original filesize for text-based ontology serialisations between 1 kilobyte and more than 500 megabyte. A very important consideration here is that Java provides built-in functionality to deal with archives of this kind, which is a strong argument for users of the OWL API (Java-based), the most comprehensive library to deal with OWL 2 ontologies for the community.

As for the entry point, we suggest a simple file naming convention. Although not really an applicable label in the case of a mere monolithic ontology, *root.owl* might be a generic name to indicate for users of the archive: this is where you start. *root.owl* might be empty or non-empty and importing its dependencies through owl:imports.

Lastly, we need to specify a standard behaviour for how to deal with imports in the case that imported ontologies are included files in the same directory structure, but referenced as URLs in the owl:imports statements. The problem is that in practice, some of the URLs are dereferenceable, some not. We propose to interpret the existence of files corresponding to the path/filename in the URL as corresponding to the intention of the distributor to prefer the local versions, as opposed to them being there merely as fallback.

The full protocol of dealing with an OWL/ZIP file is then as follows:

1. Unzip the archive
2. Find the root.owl entry point
3. For each importing, attempt to resolve imports locally and only on failure to attempt to deference the imported URL from the web

4 Use case: the Atomic Decomposition in OWL/ZIP

Reasoners that make use of modular reasoning techniques such as Chainsaw [7] might benefit from an ontology that was decomposed at deploy time. We have developed our own method to represent an Atomic Decomposition (AD) as a set of ontology files where the dependencies are represented as owl:imports (compare Figure 1). The Atomic Decomposition can be seen as a terse representation of the modular structure of an ontology, represented by nodes and edges.

Each node represents an atom (a set of axioms), and each edge represents a directed dependency relation to another atom. An atom including all its dependencies corresponds to a genuine module. For the technical details of the AD the reader is referred elsewhere [8]. To make optimal use of the Atomic Decomposition, we represent connected components of the Atomic Decomposition dependency graph, the minimal (atoms without further dependencies) and the maximal (atoms without further dependents) atoms by using file naming conventions. The root ontology imports the set of connected components and the tautologies of the original ontology (which would otherwise be lost). The connected components (not all nodes in the AD are necessarily connected) import their respective maximal atoms, which in turn import other atoms. Minimal atoms are the leaf nodes of the dependency graph and do not import anything. Some results with respect to the serialisation of the Atomic Decomposition are presented in Section 6.2.

Thus, we show how one might encode additional useful information onto of the base OWL/ZIP format. Such conventions can ease the building of layered tools.

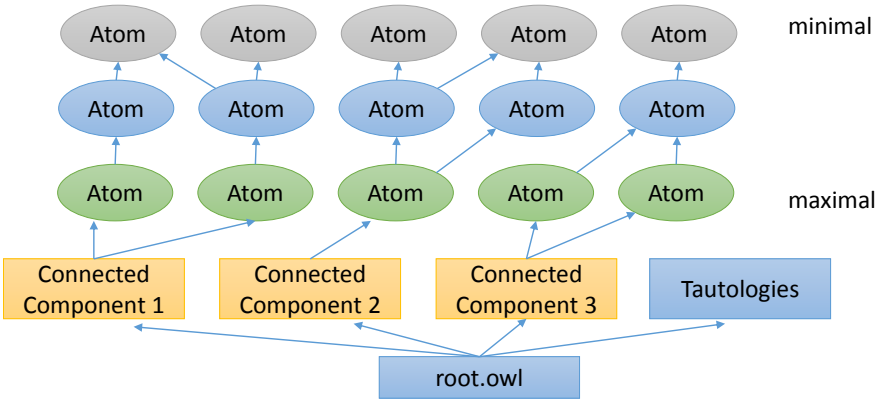

Fig. 1. Simplified structure of the serialised Atomic Decomposition. Edges represent owl:imports statements.

5 OWL/ZIP for monolithic ontologies

5.1 Experimental Setup

We have conducted a survey using the ORE 2014 dataset [5], largely based on a cleaning strategy described in [4], on a standard laptop, Lenovo Thinkpad T520, Windows 7 Professional 64-bit, Memory: 15.9 GB (1333MHz, DDR3), Intel(R) Core(TM) i7-2760QM CPU @ 2.40GHz X 8. All ontologies in the corpus

come with imports closure merged into the root ontology (monolithic). For every ontology in the corpus, we loaded it with the OWL API (version 3.5.0), zipped it, unzipped it and loaded it again, and timed the various steps (wall-clock time, using System.currentTimeMillis() in Java). Each ontology was processed in a separate Java process (Virtual Machine).

5.2 Results

Table 1. Summary statistics for using OWL/ZIP with monolithic ontologies.

	Min()	Max()	Mean()	Median()
Compression Rate	81.65%	98.59%	93.30%	93.13%
Added Load	3.27%	93.47%	24.78%	23.26%

Compression rates across corpus: As can be seen in Table 1, compression rates can reduce the file size by at least around 80%, up to a maximum of almost 99%. A closer look at Figure 2 reveals that the effects of the compression (unsurprisingly) are getting better for larger ontologies. Ontologies with more than 1000 logical axioms generally get compression rates above 80%, tending towards 90% and more in very large ontologies above 90,000 axioms. Worse compression rates of less than 85% are generally found only among the very small ontologies (1000 or less logical axioms).

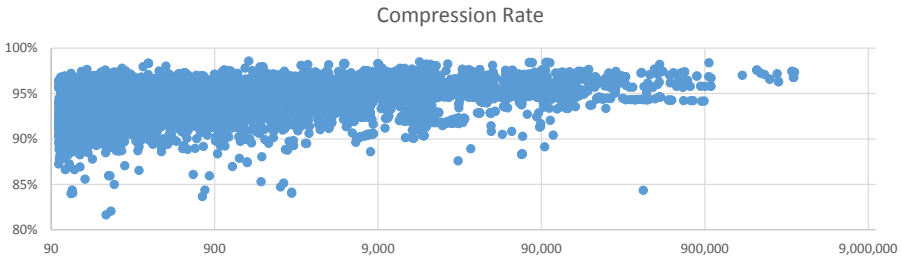

Fig. 2. Compression rates (y-axis) against ontology size (x-axis), measured in logical axiom counts.

Decompression overhead: Figure 3 shows the load time overhead for the decompression. The x-axis shows the original load time in seconds, the y-axis the load time overhead induced by the decompression in percent. The overhead is never more than 100%, i.e., if an ontology in its uncompressed form (x-axis) would take one second to load, the total load time would never be more than 2 seconds. Bad overheads of 80%-100% however only occur in the in the 100

millisecond (or less) area. The longer the ontologies would take to load in their uncompressed form, the less effect does the decompression time have on the overall loading time. For ontologies that take more than 10 seconds to load, the overhead is never more than 50%.

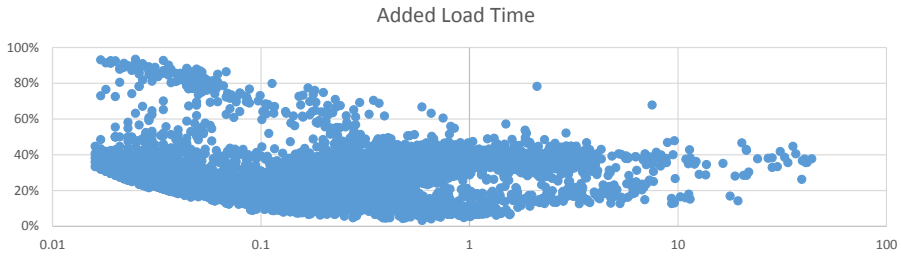

Fig. 3. Added load time (y-axis) relative to original load time in seconds (x-axis).

6 OWL/ZIP for modular ontologies

One of the benefits of using zip archives is the possibility to deploy an ontology in its modularised form. We have conducted a short survey to see how the use of OWL/ZIP might benefit or limit deploying such an ontology. We are investigating a rather extreme case, were we deploy the ontology as its Atomic Decomposition (see Section 4). These kinds of decompositions are extreme in the sense that they contain a lot of files (i.e., each atom will be serialised as a separate file on the file system), many more than a manual decomposition would ever contain. We used the $\bot$-decomposition [8].

6.1 Experimental Setup

For the general setup see Section 5.1. We randomly sampled from the same corpus 10 small (100-1000 logical axioms), 10 medium sized (1000-10000 log. ax.), 10 large (10000-100000 log. ax.) and 10 very large ontologies (¿100000 log. ax.), serialised them into their decomposed from as described in Section 4. We introduced a sharp timeout of 30 minutes for the entire process of loading, decomposing, serialising, zipping, unzipping and loading again.

6.2 Results

Out of the 40 ontologies in the testset, 23 were processable within the given timeout. *Compression rates:* As can be seen in Figure 4 (right) and Table 2, the effect of the compression is relatively stable across the testset, generally between

60% and 80% (with a single outlier in the better end). The figure shows the compression rate (y-axis) with respect to the number of atoms in the decomposition. This number serves as a proxy for the size of an ontology; the number of atoms is strongly correlated with the number of axioms in the type of decomposition we used here. We use atoms here instead of axioms because it gives a better sense into how many files the ontology is split. *Decompression overhead:* Even more clearly than in the case of monolithic ontologies, we can observe that the bigger the original load time, the smaller the relative impact of the decompression. Figure 4 (left) shows the load time for the uncompressed ontology (x-axis) in seconds with respect to the relative added load through the decompression. The larger the load time of the ontology in its uncompressed form, the smaller the relative impact of the unzipping on the total load time. The worst case in the test set imposed an additional 56% on the load time, which seems reasonable (Table 2).

Table 2. Summary statistics for using OWL/ZIP with modular ontologies.

	Min()	Max()	Mean()	Median()
Compression Rate	66.18%	92.28%	71.41%	70.59%
Added Load	2.79%	56.11%	30.67%	30.21%

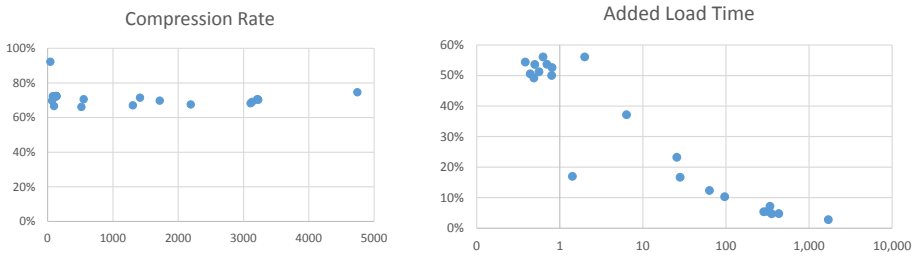

Fig. 4. Left: added load time (relative, y-axis) with respect to original load time in seconds (x-axis, log-scale), Right: compression rate (y-axis) with respect to numbers of atoms in the decomposition (x-axis)

7 Discussion

The results indicate that using a compressed form to deploy ontologies only imposes a reasonable degree of performance reduction on the load time. Furthermore, the general compression rates are compelling at least for the larger ontologies. The differences between the monolithic and the modular cases are

most likely due to the overhead induced by having many, potentially thousands of separate files (file listings in the archive header, etc).

It is possible that with a more streamlined implementation (i.e. rather than uncompressing to disk and then loading via normal mechanisms, we would directly uncompress to the parser in memory) that the total load (or save) time could be *reduced* relative to the uncompressed process. This could happen for ontologies which are large enough that IO costs dominate and we have sufficient RAM. (If you consider loading an ontology from the web it's clear that in many cases download time dominates by a large factor.)

If we compare our current results with a recent binary OWL proposal [3], we see that we have better compression (5x to 10x vs. 4x) and much worse parse times (i.e., slow downs vs. dramatic speed ups). We first note that their experiments were on a much smaller and idiosyncratic corpus, but also that they may be missing some easy compression wins (e.g., they replace uses of IRIs with indexes, but they, as far as we can tell, do not compress the original set of IRIs; considering the typical amount of shared prefixes in IRIs in an ontology, it is obvious that significant gains could be made). However, a simple implementation of a binary OWL parser is considerably more challenging than of an OWL/ZIP one and OWL/ZIP files are easily inspected by hand using ubiquitous tools (i.e., a ZIP decompressor and a text editor).

While more accessible than binary formats, OWL/ZIP still imposes some overhead. Until existing editors catch up, users have to manually unzip and manage the resulting directory. Our reuse of imports helps to a certain extent, but even there, existing OWL IDEs are designed to handle hand-built imports structures.

The zipped directory allows for further easy extension. For example, reasoners wishing to cache some state between invocations can put that into a subdirectory with some hope that the compression would mitigate the size hit and that the user could be safely unaware of this extra information.

8 Conclusions and Future Work

We have proposed a convention to distribute large monolithic and modular ontologies in a compressed form (OWL/ZIP). The approach covers both an unambiguous interpretation of the imports closure and a proposal of a specific format for the compression. The general rationale for such a format is the sheer size of many ontologies and the need to distribute decompositions. The results of our survey indicates that using a standard form of compression like zip is sufficient to deal with this problem, and there are many tools and APIs available that can deal with it. An interesting investigation would be to include an OWL/ZIP parser in the OWL API that allows to unzip in memory, using the Java 7 FileSystem class. This could potentially give benefits at load time, even over the standard parsing, due to possible reductions in IO touches. But at least it should make dealing with compressed ontologies faster.

References

1. E. Antezana, M. Egaña, W. Blondé, A. Illarramendi, I. n. Bilbao, B. De Baets, R. Stevens, V. Mironov, and M. Kuiper. The Cell Cycle Ontology: an application ontology for the representation and integrated analysis of the cell cycle process. *Genome biology*, 10(5):R58, Jan. 2009.
2. C. Golbreich, J. Grosjean, and S. J. Darmoni. The Foundational Model of Anatomy in OWL 2 and its use. *Artificial intelligence in medicine*, 57(2):119–32, Mar. 2013.
3. M. Horridge, T. Redmond, T. Tudorache, and M. A. Musen. Binary OWL. In *OWLED*, 2013.
4. N. Matentzoglu, S. Bail, and B. Parsia. A Snapshot of the OWL Web. In *International Semantic Web Conference (1)*, pages 331–346, 2013.
5. N. Matentzoglu and B. Parsia. ORE 2014 Reasoner Competition Dataset. http://zenodo.org/record/10791, July 2014. Dataset, Accessed: 2014-07-30.
6. N. F. Noy, N. H. Shah, P. L. Whetzel, B. Dai, M. Dorf, N. Griffith, C. Jonquet, D. L. Rubin, M. A. Storey, C. G. Chute, and M. A. Musen. BioPortal: Ontologies and integrated data resources at the click of a mouse. *Nucleic Acids Research*, 37, 2009.
7. D. Tsarkov and I. Palmisano. Chainsaw: A metareasoner for large ontologies. In *OWL Reasoner Evaluation Workshop (ORE 2012)*, 2012.
8. C. D. Vescovo, B. Parsia, U. Sattler, and T. Schneider. The Modular Structure of an Ontology: Atomic Decomposition. In *IJCAI*, pages 2232–2237, 2011.