ICBO: International Conference on Biomedical Ontology
July 28-30, 2011 · Buffalo, NY, USA

owl_cpp, a C++ Library for Working with OWL Ontologies

Mikhail K. Levin[1], Alan Ruttenberg[2,3], Anna Maria Masci[4], Lindsay G. Cowell[1]

[1]Department of Clinical Sciences, University of Texas Southwestern Medical Center at Dallas, TX, USA
[2]School of Dental Medicine, University at Buffalo, NY, USA
[3]Science Commons, Mountain View, CA, USA
[4]Department of Biostatistics and Bioinformatics, Duke University Medical Center, Durham, NC, USA

Abstract. Here we present *owl_cpp* (http://sf.net/projects/owl-cpp/), an open-source C++ library for parsing, querying, and reasoning with OWL 2 ontologies. *owl_cpp* uses Raptor, FaCT++, and Boost libraries. It is written in standard C++, and therefore can be used on most platforms. *owl_cpp* performs strict parsing and detects errors ignored by other parsers. Other advantages of the library are high performance and a compact in-memory representation.

1 Introduction

The OWL Web Ontology Language, one of the Semantic Web technologies, is designed to formally represent human knowledge and facilitate its computational analysis and interpretation. OWL and other Semantic Web technologies have been successfully applied in many areas of biomedicine. Their success is in part due to a broad spectrum of software developed in support of the Semantic Web.

Working with ontologies involves several common tasks, such as parsing OWL documents, querying their in-memory representation, passing information to a Description Logic (DL) reasoner, and executing DL queries. In computer memory ontologies are represented either as RDF triples or as axioms and annotations. A triple is a simple statement consisting of three nodes, where the predicate node expresses a relationship between the subject and object nodes. Although a set of triples can represent a complex graph of interrelated entities, in computer memory it is stored as a uniform array that can be efficiently searched and queried. Therefore, triple stores are used when one needs to efficiently perform relatively simple queries. Axioms are statements about ontological classes and instances. Each axiom may be seen as a graph corresponding to one or many RDF triples. Since axioms are more complex than triples, querying them is less efficient. However, with the help of a reasoner, one can execute more sophisticated DL queries and discover knowledge implicitly contained in the ontology.

Despite the availability of highly developed tools, working with large biomedical ontologies remains challenging. Some of the problems we face are *a*) detection and elimination of errors in OWL documents; *b*) absence of Java virtual machine (JVM) support on some high-performance computing (HPC) platforms; *c*) limited availability of semantic web tools in programming languages such as C++, Python and Perl; *d*) a large footprint of ontology in-memory representation; and *e*) poor parsing and querying performance.

To address these problems we developed owl cpp, a library for parsing, querying, and reasoning with ontologies. Its key features include *a*) strict parsing, detecting errors in OWL documents; *b*) written in standard C++, can be compiled on most platforms; *c*) requires no virtual machine; *d*) possibility of creating efficient APIs for other languages, e.g., Java, Perl, Python; *e*) small memory footprint; and *f*) high performance.

2 Implementation

The following basic operations should be supported by *owl_cpp*: *a*) making a catalog of OWL 2 RDF/XML documents in user-supplied locations, *b*) parsing OWL 2 RDF/XML documents and their imports, *c*) storing and searching resulting RDF triples, *d*) converting the triples into axioms and loading them into a

reasoner, and *e*) performing description logic queries. This functionality should be implemented along the following guidelines:

- *Correctness and reproducibility* should be verified with extensive unit tests.
- *Strict syntactic verification* should be performed during parsing and axiom generation to prevent possible semantic errors.
- *Error messages* should contain sufficient information to correct the error.
- *Efficiency* should be maximized both in terms of speed and in terms of runtime memory footprint to be able to process large ontologies.
- *Portability* should be maximized by using only standard C++ features.
- *Maintainability* should be maximized by implementing *owl_cpp* as a decoupled modular structure and by utilizing well--established libraries, i.e., the C++ Standard Library and Boost (http://www.boost.org/).

Currently, *owl_cpp* is composed of three modules. **The parsing module** is a C++ wrapper for Raptor, a popular C library for RDF parsing [1]. To our knowledge, Raptor is the only C/C++ RDF parser under active development. To parse XML, Raptor uses the SAX interface of Libxml2 library (http://xmlsoft.org/). *owl_cpp* reads ontologies only from STL input streams and from filesystem locations specified by the user. Although, by default, Raptor may attempt to fetch ontologies from the internet, this functionality is disabled in the interests of reliability, security, and performance.

The triple store module is responsible for storing, searching, and retrieving of RDF triples. The store internally provides separate containers for namespace URIs, nodes, and triples. The containers keep track of the objects by mapping them against light-weight IDs. Retrieval of an object by such an ID is as efficient as array indexing. Each RDF triple is stored as the three IDs of its corresponding nodes. Although many mentions of the same node may be found in an ontology document, only one instance of each node is kept in the store.

The search of triples is frequently performed and therefore should be highly optimized. Although a straightforward task, it is made complicated by the number of potential configurations. Depending on the node IDs provided by the user, the triples may be searched in eight different ways: just by the subject, predicate, or object, or by any of their combinations, including the configuration, where no IDs are provided, which should return all triples in the store. Furthermore, as a result of the search, the user may be interested in three types of return values: a complete list of triples that match the provided IDs, only the first triple found, or merely a boolean indicating whether the search was successful. Since for any of the eight search configurations, any of the three types of return values may be required, the total number of possible configurations is 24. Clearly, implementing a separate method for every search configuration would unacceptably clutter the interface. On the other hand, a one-for-all-configurations method would necessarily sacrifice performance. Therefore, in keeping with "you pay only for what you use" principle, the search was implemented as a non-member template function that accepts either `Node_id` or `Blank` types for each of the three terms. The return type is a `boost::iterator_range` (http://www.boost.org/doc/libs/release/libs/range/index.html), which can be implicitly converted to a boolean or used for iterating over the matching triples.

The reasoning module functionality is performed by FaCT++, which is, to our knowledge, the only open-source C/C++ DL reasoner library [2]. *owl_cpp* passes information from the triple store to FaCT++ by converting triples to axioms. Currently the conversion is done using the Visitor design pattern [3, 4]. DL queries are currently performed directly through the FaCT++ interface with the aid of actor and predicate classes supplied by *owl_cpp*.

3 Results and Discussion

The interface of *owl_cpp* is designed to simplify basic operations with ontologies. For example, the file `ontology.owl` is read into the triple store by calling `load("ontology.owl", store)`. To load the ontology along with its imports, a catalog of ontology IDs and locations should also be provided:

```
load("ontology.owl", store, catalog).
```

Once loaded into the store, the triples can be queried. For example, expression `find_triples(blank, T_rdf_type::id(), T owl Class::id(), store)` finds all triples that declare classes. The axioms are copied from the triple store to the FaCT++ reasoning kernel by calling `add(store, kernel)`.

The accuracy of parsing and reasoning of *owl_cpp* was tested with many ontologies. Some of the smaller OWL documents (e.g., several OWL 2 Test Cases, http://owl.semantic web.org/page/OWL 2 Test Cases) were incorporated into unit tests, which assert their consistency and satisfiability. Further testing was done during the development of the Ontology of Biological Pathways [5] by executing DL queries formulated by domain experts and comparing the results with ones from Protégé (FaCT++ and HermiT reasoners). The results were always identical.

One of the advantages of *owl_cpp* over other OWL libraries is its ability to discover syntactic errors, thus preventing incorrect semantic interpretation of ontologies. During ontology development in our group, *owl_cpp* has detected inconsistent import statements, undeclared property and annotation predicates, misspelled standard OWL terms, and other problems.

Parsing performance of *owl_cpp* was tested using OpenGALEN ontologies version 8 in OWL/RDF format (http://www.opengalen.org/). The ontologies occupied 0.5 GB on the hard disk and consisted of 9.7 million triples. Their parsing was successful after correcting several errors, e.g., replacing `owl:propertyChain` terms with `owl:propertyChainAxiom`. The rate of parsing was estimated to be 108 thousand triples per second. The bottleneck of this process appeared to be the insertion of new terms into the triple store. The ways to streamline this step are currently being investigated.

Future development of *owl_cpp* includes the following tasks: *a*) defining high-level C++ APIs for parser, triple store, and reasoner; *b*) designing axiom-based API; *c*) improving readability of error messages; *d*) designing APIs for other programming languages; *e*) introducing support for other OWL 2 syntaxes, e.g., Manchester, Turtle, OWL/XML; and *f*) designing a module for batch execution of OWL 2 Test Cases.

Acknowledgments

The authors would like to thank Dmitri Tsarkov for his help with FaCT++ library. This work was supported by an NIAID-funded R01 (AI077706) and a Burroughs Wellcome Fund Career Award to LGC.

References

1. D Beckett (2001) The Design and Implementation of the Redland RDF Application Framework, in Proc. of the Tenth International World Wide Web Conference.
2. D Tsarkov, I Horrocks (2006) FaCT++ Description Logic Reasoner: System Description, Lecture Notes in Artificial Intelligence, in Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006), Lecture Notes in Artificial Intelligence (Springer), Vol. 4130, pp 292–297.
3. E Gamma, R Helm, R Johnson, J Vlissides (1995) Design Patterns (Addison-Wesley, Boston, MA).
4. A Alexandrescu (2001) Modern C++ design: generic programming and design patterns applied (Addison-Wesley, Boston, MA).
5. AM Masci, MK Levin, A Ruttenberg, LG Cowell (2011) Connecting Ontologies for the Representation of Biological Pathways, in Proc. International Conference on Biomedical Ontology.