The OWL API: A Java API for OWL Ontologies

Editor(s): Pascal Hitzler, Kno.e.sis Center, Wright State University, Dayton, USA
Solicited review(s): Jérôme Euzenat, INRIA Grenoble Rhône-Alpes, France; Peter Haase, fluid Operations, Walldorf, Germany; Frederick Maier, Kno.e.sis Center, Wright State University, Dayton, USA

Matthew Horridge [a], Sean Bechhofer [a,*]
[a] *School of Computer Science, Kilburn Building, Oxford Road, University of Manchester, M13 9PL, United Kingdom*
email: first.second@manchester.ac.uk

Abstract. We present the OWL API, a high level Application Programming Interface (API) for working with OWL ontologies. The OWL API is closely aligned with the OWL 2 structural specification. It supports parsing and rendering in the syntaxes defined in the W3C specification (Functional Syntax, RDF/XML, OWL/XML and the Manchester OWL Syntax); manipulation of ontological structures; and the use of reasoning engines. The reference implementation of the OWL API, written in Java, includes validators for the various OWL 2 profiles - OWL 2 QL, OWL 2 EL and OWL 2 RL. The OWL API has widespread usage in a variety of tools and applications.

Keywords: OWL, API, Java, reasoning, application development

1. Introduction

The Web Ontology Language OWL has been a W3C Recommendation [34] since 2004, with OWL 2 [43] refining and extending the original specification. A pre-requisite for the success and adoption of languages is the existence of tooling, and OWL is no exception, with a wide range of (free, open-source and commercial) tools now available. These tools support the creation and editing of OWL ontologies, reasoning over ontologies, and the use of ontologies in applications.

In this paper, we discuss the OWL API, a high level Application Programming Interface (API) that supports the creation and manipulation of OWL Ontologies. The OWL API has been available since 2003, and has undergone a number of design revisions, in particular tracking the evolution of OWL itself. Key aspects of the OWL API include an axiom-centric abstraction, first class change support, general purpose reasoner interfaces, validators for the various OWL 2 profiles, and support for parsing and serialising ontologies in a variety of syntaxes. The OWL API also has a flexible design that allows third parties to provide alternative implementations for all major components. The OWL API is implemented in Java and is available as open source under an LGPL licence[1].

Since its initial development in 2003 [7], the OWL API has been used in multiple development projects, including Protégé-4 [26], SWOOP [24], the NeOn Toolkit [17][2], OWLSight[3], OntoTrack [27], the Pellet reasoner [38] [4] and a number of online validation and conversion services[5]. Work on early versions of the OWL API contributed valuable implementation experience [3] to the WebOnt Working Group[6], facilitating

[*]Corresponding author

[1]`http://owlapi.sourceforge.net`
[2]`http://www.neon-toolkit.org/`
[3]`http://pellet.owldl.com/ontology-browser/`
[4]`http://clarkparsia.com/pellet`
[5]`http://owl.cs.manchester.ac.uk`
[6]`http://www.w3.org/2001/sw/WebOnt`

the production of the OWL Recommendation. Across all versions, the OWL API has over 34,000 downloads (with 500+ downloads per month over the last year), reinforcing our view that it has played, and continues to play, a key role in promoting the uptake and use of OWL.

2. OWL API Design Philosophy

The original inspiration for the OWL API came from observations of the impact made by the provision of the XML Document Object Model (DOM) [42]. The DOM, along with freely available implementations (such as the Java implementations in Sun's JDK [40]) allowed a large number of developers to use and manipulate XML in applications, which in turn facilitated the widespread adoption of XML. Our belief was that a similar effort would prove of great benefit to the adoption of OWL.

The provision of an OWL API can allow developers to work at an appropriate level of abstractions, isolating them from potential issues related to, for example, serialisation and parsing of data structures. This was a particular consideration with OWL, given OWL's close relationship with RDF [25] and its triple-based representation.

In order to provide this "high level view", the design of the OWL API is directly based on the OWL 2 Structural Specification [32]. An ontology is simply viewed as a set of axioms and annotations as depicted in Figure 1. The names and hierarchies of interfaces for entities, class expressions and axioms in the OWL API correspond closely to the structural specification, relating the high level OWL 2 specification directly to the design of the OWL API. The OWL API supports loading and saving ontologies is a variety of syntaxes. However, none of the model interfaces in the OWL API reflect, or are biased to any particular concrete syntax or model. For example, unlike other APIs such as Jena [11], or the Protégé 3.X API, the representation of class expressions and axioms is *not* at the level of RDF triples.

3. Detailed Design Principles

Although aspects of the API design have evolved since its original inception and implementation, a number of basic design principles have endured. These are outlined below, but can be summarised as:

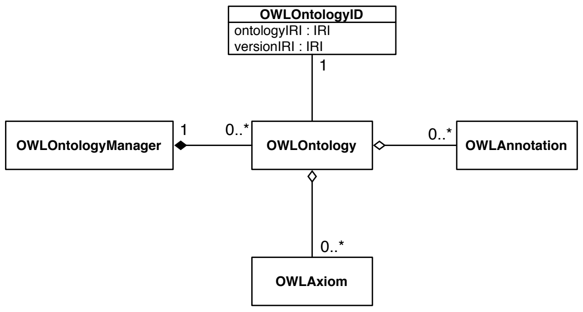

Fig. 1. A UML diagram showing the management of ontologies in the OWL API. The `OWLOntology` interface provides accessors for efficiently obtaining the axioms contained within an ontology. The method of storing axioms is provided by different implementations of the `OWLOntology` interface. The design of the API makes it possible to mix and match implementations, for example, an in-memory ontology could be used together with one that is stored in a database and one that is stored in some kind of triple store. The OWL API reference implementation provides an efficient in-memory storage solution.

– Interfaces providing read-only access to the model structure.
– Change/manipulation through explicit change operations.
– Independence from concrete serialisations (in particular triple based representations).
– A clear separation between components providing particular functionalities, such as representation, manipulation, parsing, rendering.
– Separation of assertion and inference.

3.1. Model

The OWL API's model provides access to an OWL ontology through a number of Interfaces and Class definitions. The model interfaces are *read-only*, in that they do not provide explicit functionality for change of the underlying data structures (see Section 3.2).

Following the OWL 2 Structural Specification, the model provides an *axiom-centric* view on OWL ontologies (see Figure 1), with an `OWLOntology` containing a number of `OWLAxiom` objects. Convenience methods are also available that will answer questions of containment such as "does Ontology O contain class C?", where containment is taken to mean that the ontology contains an axiom referring to C. This supports what we might call *locality of information*, in that all assertions about classes are associated with a particular ontology. As OWL operates in an open web context, different ontologies can make different assertions about the same classes and properties.

Although this “axiom-centric” view differs from the original OWL API design, which provided a “frame-oriented” model, the desire to provide a level of abstraction that insulates developers and applications from underlying syntactic presentations, in particular RDF or triple-based representations has been a cornerstone of the OWL API’s approach.

The model data structures make extensive use of the Visitor pattern [14] which allows separation of data structure from functionality. The Visitor makes it easier to add functionality to a class hierarchy (and is particularly suited to situations where data structures represent abstract syntax). The Visitor pattern does, however, have the drawback that changes to the data structure can be expensive as it may require changes to the Visitor implementations. For the OWL API, however, where the data structures are stable, the pattern is a good fit.

The specification of the data model is largely through Java Interfaces, enabling the use of alternative implementation strategies. The reference implementation provides data structures for efficient in-memory representations of ontologies. For many purposes this is sufficient. For example, recent versions of the National Cancer Institute (NCI) ontology can comfortably fit into around 700MB of memory (100MB of memory without annotations). However, the API has been designed so that it is possible to provide other storage mechanisms for ontologies such as relational databases or triple stores and to “mix and match” storage implementations, so that an ontology imports closure could contain in-memory representations of ontologies, ontologies persisted in secondary storage in the form of a custom database, and ontologies stored in triple stores.

While the API does not include these alternative storage mechanisms out of the box, third parties have developed such solutions. An exemplar solution, called OWLDB, has been developed by Kleb et al [21]. Their solution stores ontologies in a relational database, using a “native” mapping of OWL constructs (as opposed to a mapping to triples) to a custom database schema. A similar approach is taken by Redmond in [36].

3.1.1. Ontology Management

Besides the model interfaces for representing entities, class expressions and axioms, it is necessary to allow applications to manage ontologies. Figure 1 shows a high level overview of this picture. The `OWLOntology` interface provides a point for accessing the axioms contained in an ontology. As discussed above, different implementations of the `OWLOntology` interface can provide different storage mechanisms for ontologies.

The `OWLOntologyManager` provides a central point for creating, loading, changing and saving ontologies, which are instances of the `OWLOntology` interface. Each ontology is created or loaded by an ontology manager. Each instance of an ontology is unique to a particular manager, and all changes to an ontology are applied via its manager.

This centralised management design allows client applications to have a single access point to ontologies, to provide redirection mechanisms and other customisations for loading ontologies, and allows client applications to monitor all changes that are applied to any loaded ontologies. The manager also hides much of the complexity associated with choosing the appropriate parsers and renderers for loading and saving ontologies.

3.2. Change

The core Ontology model interfaces are *read-only*, with issues of change (additions and removals of axioms) handled through the use of explicit change objects. Class `OWLOntologyChange` provides a top level change object with further subclasses defined that encapsulate particular changes. Change enactment is then provided through a further use of the Visitor pattern, with `OWLOntologyChangeVisitor` objects enacting changes.

This use of explicit change objects follows the Command design pattern [14] which encapsulates a change request as an object. Changes are then enacted by a Visitor object.

This use of the Command pattern facilitates support for operations such as undo or redo, and the encapsulation of changes as operations provides a mechanism with which to track changes and support version management. Change objects also provide a convenient place for storing metadata about the changes, for example the user who requested the change – information which is again crucial in supporting the ontology management and editing process.

All ontology changes are applied through an ontology manager. This means that it is possible to apply a list of ontology changes, which make multiple changes to multiple ontologies, as a single unit. This works well for applications such as ontology editors, where an edit operation such as entity name change (entity IRI change) can involve the addition and removal of multiple axioms from multiple ontologies—it is possible to group the changes together to form one "editor operation" and apply these changes at one time.

3.3. Inference and Reasoning

A key aspect of OWL is the provision of a semantics that defines precisely what entailment means with respect to OWL ontologies, and provides formal descriptions of properties such as consistency. The implementation of these semantics is a non-trivial matter. By separating reasoning functionality, we can relieve implementors of the burden of this, while allowing those who do provide such implementations to be explicit about this in their advertised functionality. In addition, as is discussed in some detail in [7], separation of assertion and inference is important in the implementation of OWL, particularly when developing user applications, as users may need explicit indication as to why, for example, hierarchical relationships are present.

The `OWLReasoner` interface provides access to functionality relating to the process of *reasoning* with OWL ontologies, supporting tasks such as consistency checking, computation of class or property hierarchies and axiom entailment. Of course, providing method signatures does not go all the way to advertising the functionality of an implementation – there is no guarantee that a component implementing the interface necessarily implements the semantics correctly. However, signatures and extensive detailed documentation of reasoner interfaces go some way towards providing an expectation of the operations that are being supported. From a client perspective, the benefits of having a well defined reasoner interface cannot be overstated. The ability to exchange one reasoner implementation for another in a client application, and know that, in terms of query answering, each reasoner implementation should exhibit the same behaviour is a major advantage. Collections of test data (such as the OWL Test Cases [10]) can allow systematic testing and a level of confidence as to whether the implementation is, in fact, performing correctly.

The reasoner interfaces have been designed so as to enable reasoners to expose functionality that provides *incremental* reasoning support. The OWL API allows a reasoner to be initialised so that it listens for ontology changes and either immediately processes the changes or queues them in a buffer which can later be processed. This design caters for the scenario where a reasoner is used within an ontology editor and, at some point, must respond when prompted to edits of the ontology, or situations where a reasoner should respond to ontology changes as they arrive.

A number of existing implementations including the CEL [1], FaCT++ [41], HermiT [31], Pellet [38], and Racer Pro [16] and SnoRocket reasoners[7] provide OWL API wrappers. These reasoning engines are thus easily integrated into OWL API based applications such as Protégé-4 or the NeOn Toolkit.

3.4. Querying

Out of the box, the OWL API does not provide any specialised ontology query answering components such as functionality for answering conjunctive queries, or SPARQL based queries. The implementation and maintenance of such components is beyond the scope of the OWL API reference implementation. However, the `OWLReasoner` interface goes a long way in providing basic query support that meets the needs of many client applications. For example, using an implementation of the `OWLReasoner` interface, a client may query the computed class and property hierarchies, determine the instances and sub/super classes of complex class expressions, and can determine the property characteristics of object and data properties. All of this query functionality is based around entailment checking functionality, which is also directly exposed via the reasoner interface. In the same way that the OWL API has pushed towards pseudo standardisation of OWL reasoner functionality through the definition of well known reasoner interfaces, it may well be the case that a future version of the API will provide "standard" query interfaces for processing conjunctive queries and SPARQL queries.

3.5. Parsing and Rendering OWL Ontologies

A benefit of aligning the OWL API with the OWL 2 structural specification is that there is no commitment to a particular concrete syntax. Conforming [39] OWL implementations or tools *must* support RDF/XML, but there are several other syntaxes that exist which are optimised for different purposes. For example, Turtle syntax [8] provides a "compact and natural text form" of RDF serialisation. The Manchester OWL Syntax [18] provides a human readable serialisation for OWL ontologies.

[7]`http://research.ict.csiro.au/software/snorocket`

The OWL API includes out of the box support for reading and writing ontologies in several syntaxes, including RDF/XML, Turtle, OWL/XML, OWL Functional Syntax, The Manchester OWL Syntax, KRSS Syntax[8] and the OBO flat file format[9]. Due to the underlying design of the API, it is possible for the imports closure of an ontology to contain ontologies that were parsed from ontology documents written in different syntaxes.

The reference implementation of the OWL API uses a registry of parsers and renderers, supporting the addition of custom syntaxes. The appropriate parser is *automatically selected* at runtime when an ontology is loaded. By default, ontologies are saved back into the format from which they were parsed, but it is possible to override this in order to perform syntax conversion tasks and "save as" operations in editors for example.

3.6. Axioms versus Triples

APIs for dealing with OWL are generally based on a high level API that allows client code to abstract away from the potentially messy details of different serialisation syntaxes. The OWL API, however, takes abstraction to the level of axioms, which is a somewhat higher level of abstraction than a triple based abstraction provided by APIs such as Jena. Aside from the fact that there are non-triple based serialisation syntaxes that can easily be handled by the OWL API, there are good reasons for abstracting away from triples: (1) The OWL 2 Specification itself is specified at the level of axioms. The OWL API is thus closely aligned with this specification; (2) It is arguable that it is more efficient and less error prone to deal with OWL ontologies at the level of axioms, rather than at the level of triples. In particular, applying changes to ontologies or finding the usage of a given entity within an ontology can be clearer in client code. For example, consider an annotated property assertion axiom:

```
ObjectPropertyAssertion(
    Annotation(rdfs:comment ``Added by MH'')
    :worksWith :Matthew :Sean)
```

This axiom is a single object in the API. It can be added to or removed from an ontology in a single operation. In a triple based representation, this simple axiom requires *six* triples to represent it. Adding or removing the axiom therefore requires the complete and correct identification and manipulation of the triples that represent this axiom. These kinds of problems are exacerbated when axioms contain complex class expressions, which may require many complex patterns of triples in their triple based representations, including those necessary to represent lists of objects. In essence, in order to maintain a valid OWL 2 ontology, client code must enforce that the addition and removal of triples must correspond to adding and removing those triples that correspond exactly to axioms; (3) Many operations, such as reading and comparing fragments of ontologies only make sense at the axiomatic level. For example, reasoner implementers tend to care about axioms—the semantics of OWL 2 DL is not defined at the triple level. Similarly, structurally comparing two ontologies or sets of axioms can be difficult and error prone if they are represented as triples. One reason for this is because the OWL 2 structural specification treats many collections as sets, so ordering is not important when it comes to comparisons. For example, the classes in a `DisjointClasses` axiom are essentially unordered. However, many collections are represented as RDF lists, which compare different for differently ordered items.

4. Evolution

Three versions of the OWL API have been developed. The first version [7] drew on earlier experiences in the implementation of OilEd [5], one of the first user tools to use Description Logic reasoning services (through use of the DIG protocol) to assist users in the construction of OIL and DAML+OIL ontologies. Although OilEd was widely used (with over 10,000 download requests), the underlying implementation suffered from problems, not least of which was the lack of a separation between interface and implementation. The first version of the OWL API addressed a number of these concerns, while carrying forward other design decisions such as extensive use of the Visitor pattern [14].

A key change in Version 2 of the OWL API was the introduction of Java Generics[10], which enabled tighter control over method typing, ensuring that, for example the result of a query for equivalent classes is returned as a set of class expressions. Version 2 also

[8]`http://dl.kr.org/krss-spec.ps`

[9]`http://www.geneontology.org/GO.format.obo-1_2.shtml`

[10]`http://download.oracle.com/javase/1.5.0/docs/guide/language/generics.html`

tracked proposed changes in OWL (known at that point as OWL 1.1 [35]), arising from the work of the OWL Working Group including the addition of qualified cardinality restrictions. In particular, there was a shift from frame-based data structures to an axiom oriented model.

Version 3, the latest release, sees an alignment of method and class names with those used in the OWL 2 Structural Specification[32]. This potentially introduces backwards compatibility issues with earlier releases of the OWL API, but our decision was that alignment with the specification brought sufficient benefits to outweigh these. For example, the W3C specifications can now themselves serve as additional documentation for the code-base. After reading the OWL 2 specification documents, users should be able to easily identify the correspondence between the OWL API and the specification. Although the name changes are largely syntactic, allowing the use of find and replace scripts to address most incompatibilities, one area of substantive change is in the modelling of axiom annotations, which has been changed to keep in line with the OWL 2 specification. In particular axiom annotations are now *embedded* axioms, thus having an effect on structural identity. In keeping with the OWL 2 Recommendation, entities in OWL ontologies are now identified using IRIs.

5. Additional Functionality

Here we discuss some of the enhanced functionality that is available in the core OWL API download supporting tasks such as profile validation, explanation and modularity.

5.1. Profile Validation

The OWL 2 specification defines OWL profiles that correspond to syntactic subsets of the OWL 2 language. The profiles are defined in the OWL 2 profiles document, namely OWL 2 EL, OWL 2 QL and OWL 2 RL. Each profile is designed to trade some expressive power for efficiency of reasoning. For example, the OWL 2 EL profile trades expressivity for the benefit of polynomial time subsumption testing. Similarly, reasoning for the OWL 2 RL profile can be implemented using a rule engine.

For those using these profiles, and tools that support them, it can be necessary to determine whether or not an ontology falls into one of the profiles or not. The OWL API contains an API to deal with ontology profiles. Various profile related interfaces are available that provide functionality to ask whether an ontology is within a profile. When doing this, a profile report is generated that specifies whether an ontology and its imports closure fall into a given profile, and if not details *why* this is the case. The profile API allows complete programmatic access by client software, with fine-grained objects that represent specific reasons for profile violations.

An OWL API based Web based application that performs profile validation on an ontology and its imports closure is available online[11]. Each item in the detailed validation report can be accessed programmatically, allowing client software to customise report rendering, or offer more advanced functionality such as repair suggestions that would take an ontology back into the desired profile.

5.2. Explanation

In recent years, explanation of entailments has been a “hot topic”. Many of the afore mentioned ontology development environments now feature the ability to generate explanations for unwanted or surprising entailments that arise during editing or browsing ontologies. Many of these explanation facilities are based on justifications [37,2,23], which are minimal sets of axioms that are sufficient for an entailment to hold. The OWL API ships with code for generating these explanations, making it easy to add basic explanation services to client applications.

5.3. Modularity

Many applications require the ability to work with well defined portions of an ontology, that are usually called modules. The OWL API contains basic functionality for extracting syntactic-locality based modules [12] from ontologies. These kinds of modules are based on the notion of conservative-extensions [15], and guarantee to preserve entailments from an ontology over a given signature.

6. Other Frameworks

There have been a number of similar initiatives to provide application interfaces aimed at OWL. HP’s

[11]`http://owl.cs.manchester.ac.uk/validator`

Jena Toolkit [11] supplies ontology interfaces that provides convenience wrappers around RDF interfaces. Comparisons of the OWL API and Jena's triple-based approach for some tasks are explored in [4].

The KAON [9] toolkit is an open-source ontology management infrastructure targeted for business applications[12]. The KAON toolkit includes an API for dealing with RDF graphs and the KAON ontology language, which is a proprietary extension of RDFS. The APIs in the KAON tool suite therefore differ from the OWL API in that KAON does not provide support for dealing with OWL or OWL 2 ontologies, and the OWL API does not provide direct support for dealing with RDF Graphs.

KAON2 [30] is the successor to KAON, and is an infrastructure for managing OWL-DL, SWRL, and F-Logic ontologies. The ontology model supported in KAON2 is less expressive than that described by OWL 2 since an important focus of KAON is performance reasoning on knowledge bases with simple ontologies and large A-Boxes. Despite this, many of our underlying design considerations conceptually follow the KAON2 design.

The Protégé-OWL API is the API on which Protégé-OWL [26] is based. The API is is influence by a mixture of the native Protégé frame-based API and the Jena API. In contrast to the OWL API's axiom oriented view, it therefore provides what amounts to a frame-based ontology API for OWL ontologies. Since Protégé-OWL is built on top of this API, it is mainly used by Protégé-OWL plugin developers, although it can be used for 3rd party application development.

7. Download, Takeup and Usage

The OWL API is available as open source under the LGPL licence and can be downloaded from Sourceforge[13]. Statistics taken from sourceforge shown in Figure 2 indicate a steady increase in downloads and web traffic[14]. Since version 1 was released, the OWL API has had over 34,000 downloads from the Sourceforge website. It should be noted that this count does not include copies of the API that are now distributed with widely used tools such as Protégé-4, the NeOn Toolkit, Pellet and HermiT.

[12]Available at `http://kaon.semanticweb.org`

[13]`http://owlapi.sourceforge.net`

[14]Note that the figures for project web traffic are only available from mid-2007.

Some examples of the kinds of applications and services that have been developed using the OWL API are discussed below. As the code-base is freely available, we fully expect there to be further third-party usage of the OWL API that we are not explicitly aware of.

Protégé-4 [19] is an open source OWL ontology editor that was initially designed and developed at the University of Manchester. Protégé-4 uses the OWL API to underpin all ontology management tasks, from loading and saving ontologies, to manipulating ontologies during editing, to interacting and offering a choice of OWL reasoners. Virtually all of the functionality provided by the OWL API is utilised by Protégé-4.

The **NeOn Toolkit** [2] [17] is an Eclipse based ontology development environment that was developed as part of the NeOn project. While early versions of the toolkit were written on top of KAON2[15], in 2010, the project moved to the OWL API[16].

OWLSight[3] is a web based ontology browser written by Clark & Parsia that uses the Pellet reasoner. The browser is written using the Google Web Toolkit, with the OWL API being used to read and access ontologies.

The **Ontology Browser**[17] dynamically generates documentation for ontologies and is based on the OWLDoc software. The OWL API is used for loading and accessing ontologies and interfacing with the FaCT++ reasoner.

OntoTrack [27] is a browsing and editing tool for OWL ontologies that is developed at Ulm University. The OWL API is used for loading and accessing ontologies for rending into a graph.

The **SKOS API**[18] [22] is a Java interface and implementation for SKOS [29], built on top of the OWL API. It adopts a similar approach to the OWL API, providing an abstract data model for SKOS that avoids commitment to particular concrete syntaxes. SKOSEd, a SKOS editor is implemented as a plugin in to Protégé-4.

The **OWLlink API** [33] implements the OWLlink protocol on top of the OWL API. Besides providing an API to access remote OWLlink reasoning engines, it turns any OWL API aware reasoner into an OWLlink server. In essence OWLlink and the OWLlink API replace the DIG [6] protocol.

[15]`http://kaon2.semanticweb.org`

[16]`http://www.neon-project.org/nw/NeOn_Toolkit_-_Latest_Major_Release`

[17]`http://owl.cs.manchester.ac.uk/browser`

[18]`http://skosapi.sourceforge.net/`

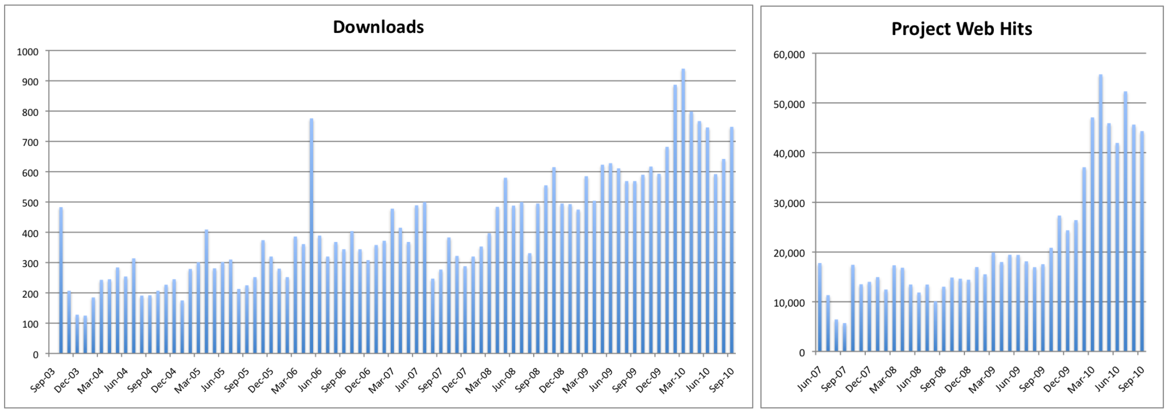

Fig. 2. Sourceforge statistics for downloads and web traffic

A number of online services[19] are implemented using the OWL API.

Syntax Converter A web based application that converts ontologies written in one OWL syntax to another OWL syntax. Syntax is converted from one format to another format by loading and saving ontologies using the OWL API.

Ontology Repository A repository of ontologies used for reasoner and tools testing. The loading and serialising makes use of the OWL API parsers and renderers, while the metrics for each ontology are computed by the OWL API's Metrics API.

Validator Uses the various profile validators to determine if an ontology and its imports closure is within a specific profile. The validator returns validation reports in a variety of human readable syntaxes. The validator makes use of the Profiles API, which is part of the OWL API.

Module Extractor Allows locality based modules to be extracted from ontologies. The extractor makes use of OWL API modularisation code, which currently provides Syntactic Locality Based Modules [20].

Metrics Allows ontologies to be submitted and returns a report about the number and types of axioms in an ontology and its imports closure. This application uses the Metrics API, which is part of the OWL API.

8. Limitations

As discussed above, the OWL API has proved to be a popular and useful code library. The design decisions made in the development of the OWL API do, however, impact on its suitability for application usage. The most crucial of these is that the OWL API is primarily designed to be an application that supports manipulation of OWL ontologies at a particular level of abstraction – which is not the RDF level. Thus the API is perhaps less suitable for those wishing to explicitly exploit the layering of OWL on RDF. In addition, the original design of the OWL API was "tuned" to target the OWL DL species. Although OWL Full ontologies can be manipulated using the OWL API, alternative frameworks such as Jena may be more suitable if OWL Full aspects are to be exploited in an application.

The reference implementation supplied with the OWL API downloads uses an in-memory representation. This places some restriction on the size of ontologies that can be processed using the OWL API. In practice, we have not found this to be a serious issue. Table 1 shows load times (in seconds) and memory consumption (in megabytes) after loading for several well known large ontologies. For comparison, the load times and memory consumption for the Jena API are also given, showing comparable performance.

9. Summary

The OWL API provides a collection of powerful and flexible interfaces supporting the use of OWL ontologies within applications. The model explicitly supports the recent OWL 2 Recommendation. Common interfaces to reasoning engines are defined, facilitating the use of inference within applications. The distribution includes a reference, main-memory implementation, and provides support for parsing and rendering ontologies in a variety of concrete syntaxes along with OWL

[19] `http://owl.cs.manchester.ac.uk`

Ontology	Axioms	Triples	Load Time	Memory	
NCI	1,093,733	1,449,558	19.0 s	638 Mb	OWLAPI
			36.3 s	600 Mb	Jena
FMA	1,651,533	1,686,457	26.5 s	831 Mb	OWLAPI
			43.8 s	482 Mb	Jena
SNOMED	1,036,800	3,355,279	30.1 s	770 Mb	OWLAPI
			59.4 s	875 Mb	Jena

Table 1

Load times and memory usage for well known large ontologies. The OWL API is compared with Jena

2 Profile validators. The API is in widespread usage, with a growing user community.

The overview presented here is necessarily brief – space constraints prevent us from providing in depth descriptions of the design and details of the code base. Further detailed documentation is available from the OWL API Sourceforge site[13] along with tutorial presentations, examples and full `javadoc` documentation for the code base. There is also an active mailing list for developers[20].

10. Acknowledgements

The OWL API has required extensive programming and design effort, and the level of sophistication and robustness achieved would have been difficult, if not impossible, without key contributions from a number of people. The authors would like to acknowledge those who have contributed to the OWL API since the first version was designed and developed by Sean Bechhofer, Raphael Volz and Philip Lord in 2003 [7]. In particular, those who have made significant contributions (in alphabetical order) include: Ron Alford[21], Nick Drummond[22], Birte Glimm[23] (reasoner interface design), Luigi Iannone[22], Aditya Kalyanpur[24], Pavel Klinov[22], Boris Motik[23] (reasoner interface and general design), Olaf Noppens[25] (reasoner interface design and parsing contributions), Ignazio Palmisano[22] (concurrency support), Timothy Redmond[26], Angus Roberts[22], Nicolas Rouquette[27], Thomas Schneider[22] (modularity support), Evren Sirin[21] (reasoner interface design, explanation code), Mike Smith[21], and Daniele Turi[22]

Thanks also go to past and present members of the Information Management Group (IMG) and the Bio-Health Informatics Group (BHIG) at the University of Manchester, and all of the OWL API users who have posted bug reports and feedback on the OWL API public mailing list.

Development of the OWL API has been funded from a number of sources, including the EU funded WonderWeb project[28]; the JISC funded CO-ODE project[29] and the EU funded TONES project[30].

[20]`http://sourceforge.net/mailarchive/forum.php?forum_name=owlapi-developer`

[21]Clark & Parsia

[22]The University of Manchester

[23]Oxford University

[24]IBM TJ Watson Research Center, New York

[25]Ulm University

[26]Stanford University

[27]JPL, NASA

References

[1] F. Baader, C. Lutz, and B. Suntisrivaraporn. CEL—a polynomial-time reasoner for life science ontologies. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 287–291. Springer-Verlag, 2006.

[2] Franz Baader and Bernhard Hollunder. Embedding defaults into terminological knowledge representation formalisms. *Journal of Automated Reasoning*, 14(1):149–180, 1995.

[3] Sean Bechhofer. OWL Web Ontology Language Parsing OWL in RDF/XML. W3C Working Group Note, World Wide Web Consortium, January 2004. `http://www.w3.org/TR/owl-parsing`.

[4] Sean Bechhofer and Jeremy J. Carroll. Parsing OWL DL: Trees or Triples? In *Proceedings of the World Wide Web Conference, WWW2004*, pages 266–275. ACM Press, 2004.

[5] Sean Bechhofer, Ian Horrocks, Carole Goble, and Robert Stevens. OilEd: a Reason-able Ontology Editor for the Semantic Web. In *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*, volume 2174 of *LNAI*, pages 396–408. Springer-Verlage, Sep 2001.

[28]`http://wonderweb.semanticweb.org/` EU IST-2201-33052

[29]`http://www.co-ode.org`

[30]`http://www.tonesproject.org/` FP6-7603

[6] Sean Bechhofer, Ralf Möller, and Peter Crowther. The DIG Description Logic Interface. In *Proceedings of DL2003 International Workshop on Description Logics*, September 2003.

[7] Sean Bechhofer, Raphael Volz, and Phillip Lord. Cooking the Semantic Web with the OWL API. In Fensel et al. [13].

[8] David Beckett and Tim Berners-Lee. Turtle - Terse RDF Triple Language. Team Submission, World Wide Web Consortium, 2008. `http://www.w3.org/TeamSubmission/turtle/`.

[9] Erol Bozsak, Marc Ehrig, Siegfried Handschuh, Andreas Hotho, Alexander Maedche, Boris Motik, Daniel Oberle, Christoph Schmitz, Steffen Staab, Ljiljana Stojanovic, Nenad Stojanovic, Rudi Studer, Gerd Stumme, York Sure, Julien Tane, Raphael Volz, and Valentin Zacharias. KAON - Towards a Large Scale Semantic Web. In K. Bauknecht, A. Min Tjoa, and G. Quirchmayr, editors, *EC-Web 2002*, volume 2455 of *Lecture Notes in Computer Science*, pages 304–313. Springer, September 2002.

[10] J. Carroll and J. De Roo. OWL Web Ontology Language Test Cases. W3C Recommendation, World Wide Web Consortium, 2004. `http://www.w3.org/TR/owl-test/`.

[11] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations. In Stuart Feldman, Mike Uretsky, Mark Najork, and Craig Wills, editors, *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83, New York, NY, USA, May 2004. ACM.

[12] Bernardo Cuenca Grau, Ian Horrocks, Yevgeny Kazakov, and Ulrike Sattler. Just the right amount: Extracting modules from ontologies. In *WWW 2007, Proceedings of the 16th International World Wide Web Conference, Banff, Canada, May 8-12, 2007*, pages 717–727, 2007.

[13] Dieter Fensel, Katia Sycara, and John Mylopoulos, editors. *Proceedings of the 2nd International Semantic Web Conference, ISWC2003*, volume 2870 of *Lecture Notes in Computer Science*. Springer, October 2003.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[15] Silvio Ghilardi, Carsten Lutz, and Frank Wolter. Did I Damage My Ontology? A Case for Conservative Extensions in Description Logic. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *The 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), Lake District, United Kingdom*. AAAI Press, June 2006.

[16] Volker Haarslev and Ralf Möller. RACER system description. In *International Joint Conference on Automated Reasoning (IJCAR 2001)*, volume 2083 of *Lecture Notes In Computer Science*, pages 701–705, 2001.

[17] Peter Haase, Holger Lewen, Rudi Studer, Duc Thanh Tran, Michael Erdmann, Mathieu d'Aquin, and Enrico Motta. The NeOn Ontology Engineering Toolkit. In Jeff Korn, editor, *WWW 2008 Developers Track*, April 2008.

[18] Matthew Horridge and Peter F. Patel-Schneider. OWL 2 Web Ontology Language Manchester Syntax. W3C Working Group Note, World Wide Web Consortium, 2009. `http://www.w3.org/TR/owl2-manchester-syntax/`.

[19] Matthew Horridge, Dmitry Tsarkov, and Timothy Redmond. Supporting early adoption of OWL 1.1 with Protégé-OWL and FaCT++. In Bernardo Cuenca Grau, Pascal Hitzler, Conor Shankey, and Evan Wallace, editors, *OWL: Experiences and Directions (OWLED)*, volume 216 of *CEUR Workshop Proceedings*. CEUR-WS.org, November 2006.

[20] Ernesto Jiménez-Ruiz, Bernardo Cuenca Grau, Ulrike Sattler, Thomas Schneider, and Raphael Berlanga Llavori. Safe and Economic Re-Use of Ontologies: A Logic-Based Methodology and Tool Support. In Sean Bechhofer, Manfred Hauswirth, Joerg Hoffmann, and Manolis Koubarakis, editors, *Proceedings of the 5th European Semantic Web Conference (ESWC 2008), Tenerife, Spain*, volume 5021 of *Lecture Notes in Computer Science*, pages 185–199. Springer, June 2008.

[21] Joachim Kleb Jörg Henss and Stephan Grimm. A database backend for OWL. In Rinke Hoeksta and Peter F. Patel-Schneider, editors, *OWL: Experiences and Directions (OWLED 2009)*, CEUR Workshop Proceedings. CEUR-WS.org, October 2009.

[22] Simon Jupp, Sean Bechhofer, and Robert Stevens. A flexible API and editor for SKOS. In *Proceedings of the 6th European Semantic Web Conference (ESWC), Heraklion, Crete*, volume 5554 of *Lecture Notes in Computer Science*, pages 506–520. Springer, 2009.

[23] Aditya Kalyanpur. *Debugging and Repair of OWL Ontologies*. PhD thesis, The Graduate School of the University of Maryland, 2006.

[24] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, Bernardo Cuenca Grau, and James Hendler. Swoop: A Web Ontology Editing Browser. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(2):144 – 153, 2006. Semantic Grid –The Convergence of Technologies.

[25] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, World Wide Web Consortium, 2004. `http://www.w3.org/TR/owl-guide/`.

[26] Holger Knublauch, Ray W. Fergerson, Natalya F. Noy, and Mark A. Musen. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In McIlraith et al. [28].

[27] Thorsten Liebig and Olaf Noppens. OntoTrack: Combining browsing and editing with reasoning and explaining for OWL Lite ontologies. In McIlraith et al. [28], pages 244–258.

[28] Sheila McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors. *ISWC 04 The International Semantic Web Conference 2004, Hiroshima, Japan*, volume 3298 of *Lecture Notes in Computer Science*, 2004.

[29] Alistair Miles and Sean Bechhofer. SKOS Simple Knowledge Organization System Reference. W3C Recommendation, World Wide Web Consortium, 2009. `http://www.w3.org/TR/skos-reference/`.

[30] Boris Motik, Alexander Maedche, and Raphael Volz. A conceptual modeling approach for building semantics-driven enterprise applications. In *Proc. of the International Conference on Ontologies, Databases and Applications of SEmantics ODBASE 2002*. Springer, LNCS, 2002.

[31] Boris Motik, Rob Shearer, and Ian Horrocks. Optimized reasoning in description logics using hypertableaux. In *Proc. of the 21st Int. Conf. on Automated Deduction (CADE-21)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 67–83. Springer, 2007.

[32] Motik, Boris and Patel-Schneider, Peter F. and Parsia, Bijan. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. W3C Recommendation, World Wide Web Consortium, 2009. `http://www.w3.org/TR/owl2-syntax/`.

[33] Olaf Noppens, Marko Luther, and Thorsten Liebig. The OWLlink api teaching owl components a common protocol. In Evren Sirin, editor, *OWL: Experiences and Directions (OWLED 2010)*, 2010.

[34] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language semantics and abstract syntax. W3C Recommendation, 10 February 2004.

[35] Peter F. Patel-Schneider and Ian Horrocks. OWL 1.1 Web Ontology Language Overview. Member Submission, World Wide Web Consortium, 2006. `http://www.w3.org/Submission/owl1-overview/`.

[36] Timothy Redmond. An open source database backend for the OWL API and protégé 4. In Evren Sirin, editor, *OWL: Experiences and Directions (OWLED 2010)*, 2010.

[37] Stefan Schlobach and Ronald Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *IJCAI International Joint Conference on Artificial Intelligence*, 2003.

[38] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2), 2007.

[39] Michael Smith, Ian Horrocks, Markus Krötzsch, and Birte Glimm. OWL 2 Web Ontology Language Conformance. W3C Recommendation, World Wide Web Consortium, 2009. `http://www.w3.org/TR/owl2-conformance/`.

[40] Sun Microsystems, Inc. Java$^{\text{TM}}$Platform. `http://java.sun.com/j2se/`.

[41] Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

[42] W3C DOM Working Group. Document Object Model. `http://www.w3.org/DOM/`.

[43] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview. W3C Recommendation, World Wide Web Consortium, 2009. `http://www.w3.org/TR/owl2-overview/`.