OWLlink

Editor(s): Bernardo Cuenca Grau, Oxford University Computing Laboratory, Oxford, UK
Solicited review(s): Ernesto Jimenez-Ruiz, Universitat Jaume I, Castelló, Spain; anonymous reviewer

Thorsten Liebig [a], Marko Luther [b], Olaf Noppens [a], and Michael Wessel [c]
[a] *derivo GmbH, James-Franck-Ring, 89081 Ulm, Germany*
E-mail: lastname@*derivo.de*
[b] *DOCOMO Euro-Labs, Landsberger Strasse 312, 80687 Munich, Germany*
E-mail: lastname@*docomolab-euro.com*
[c] *Racer Systems GmbH & Co. KG, Blumenau 50, 22089 Hamburg, Germany*
E-mail: lastname@*racer-systems.com*

Abstract. A semantic application typically is a heterogenous system of interconnected components, most notably a reasoner. OWLlink is an implementation-neutral protocol for communication between OWL 2 components, published as a W3C member submission. It specifies how to manage reasoning engines and their Knowledge Bases, how to assert axioms, and how to query inference results. A key feature of OWLlink is its extensibility, which allows the addition of required functionality to the protocol. We introduce the OWLlink structural specification as well as three bindings which use HTTP as concrete transport protocol for exchanging OWLlink messages rendered according to selected OWL 2 syntaxes. Finally, we report on existing APIs, reasoners and applications that implement OWLlink.

Keywords: OWL, OWL 2, Protocol, Reasoning, Semantic Web

1. Introduction

The W3C Web Ontology Language (OWL) is an ontology representation language with the purpose of enabling applications to meaningfully process information by reasoning. In practice, not only a concrete syntax but also a common protocol is needed to interact with components that supply reasoning services to tools or applications. OWLlink[1] is a W3C member submission which adds this piece of ontology infrastructureby providing an extensible protocol for communication among OWL 2-aware applications. The OWLlink protocol is defined by a conceptual specification [9] and comes with different bindings in terms of concrete syntaxes paired with particular transport mechanisms.

OWLlink is in the heritage of nowadays outdated protocols such as DIG [1] and KRSS [15]. It neither depends on a particular data model such as SPARQL 1.1 [5] nor on a specific transfer protocol or vendor-specific service choice as offered by the PelletServer [4]. It differs from ontology APIs, such as the Java-based OWL API [7], in that it is language-neutral and open in how to encode and transmit API calls and responses. This provides more flexibility to developers of semantic applications because they can choose from a broader range of reasoning components as well as exchange them more easily. Moreover, such a protocol based link-up typically increases the stability of the overall system. Reasoning can be transparently separated from the application and it inherently adds the option of outsourcing this task to other machines.

OWLlink consists of a protocol core and a set of bindings. The core [9] specifies how to introspect the capabilities of an OWL reasoning engine and how to set common or specific system options. It also defines primitives for dealing with Knowledge Bases (KBs), such as the creation or deletion of KBs or the successive assertion of axioms. Furthermore, the core offers

[1] `http://www.owllink.org/`

a set of basic queries to access the standard inference services supported by OWL reasoning engines.

The OWLlink bindings specify the respective transport protocols as well as the content serializations. The primary binding is XML over HTTP [14], but alternative bindings such as Functional [20] or S-Expression Syntax over HTTP [19] are available as well. New bindings can be defined as appropriate.

The OWLlink core aims at covering basic query and management functionality for typical applications. However, over time requirements may change or new systems with novel services might emerge. To allow developers to add their desired functionality, OWLlink provides an extension mechanism. The definition of new extensions is explicitly intended to be an open and community-driven process. Since there are activities aiming at widely accepted ontology query languages we expect corresponding OWLlink extensions in the near future.

This paper provides an introduction to the OWLlink protocol as of July 2010, which is fully aligned to the OWL 2 W3C recommendation from October 2009 [12] and accepted by the W3C as a member submission [9]. In the following we summarize the protocol basics and outline the core structural specification together with the extension mechanism. This is followed by a description of the set of initial bindings and a survey of supporting protocol implementations.

2. Basic protocol

OWLlink is a client-server protocol for interaction of an application (the client) with an ontology reasoning system (the server). It is designed for OWL 2 as ontology exchange language but potentially supports a wide range of logic-based languages. OWLlink does not address issues such as transactions, authentication, encryption, compression, concurrency and so on. Some of those features might be provided transparently by the access protocol (e. g., HTTP/1.1) of a particular binding or need to be added by an OWLlink extension.

This section describes the basic design of the communication protocol and introduces its structural specification. The latter defines the conceptual mechanisms for asserting axioms, asking queries and defining extensions. This structural specification makes use of a subset of the UML class diagram notation and uses the UML classes provided by the OWL 2 specification. OWLlink is in compliance with the OWL 2 conformance conditions [17] and OWLlink servers have to meet the respective tool conformance requirements. The only notable exception is, that the default language binding of OWLlink is based on the OWL 2 XML serialization (rather than RDF/XML).

2.1. Sessions, messages, and errors

An OWLlink session abstracts the actual bidirectional communication channel between the client and the server. It provides primitives to transport requests and responses. The actual implementation of a session is defined by the transport syntax and mechanism in terms of an OWLlink binding (see Section 3).

The basic interaction pattern is that of request-response. Each request is paired with exactly one response. Depending on the transport mechanism, it might be inefficient to send individual requests to a server one by one. Therefore, OWLlink requests are bundled into messages. A `RequestMessage` encapsulates a list of `Request` objects, whereas a `ResponseMessage` encapsulates a list of `Response` objects as shown in Figure 1. The server must send the responses to the client in exactly the same order in which the requests were received. If a request has been processed successfully, the type of the returned response depends on the request type. For instance, if a request addresses a specific KB, i. e. the request is a subclass of the abstract class `KBRequest`, the corresponding response has to be a subclass of the abstract class `KBResponse`. If a successful request does not produce any specific data, the server has to return a subclass of the general `Confirmation` response, e. g. an `OK`, to the client. Any `Confirmation` may carry a warning string intended to be meaningful to a human user.

If a request fails, the server has to return an `Error` response to the client containing a message reporting the cause of failure. Specific error classes allow the reporting of syntactic violations (`SyntaxError`), semantic problems (`SemanticError`) and issues regarding the management of a Knowledge Base (`KBError`). Particular errors do come with more specific errors types such as the `UnsatisfiableKBError` which is raised in response to queries referring to an unsatisfiable KB. If a server cannot process a request, it should attempt to recover gracefully, and process other pending requests as if the error did not happen. If, however, this recovery is not possible, the server should send an `Error` response and close the session.

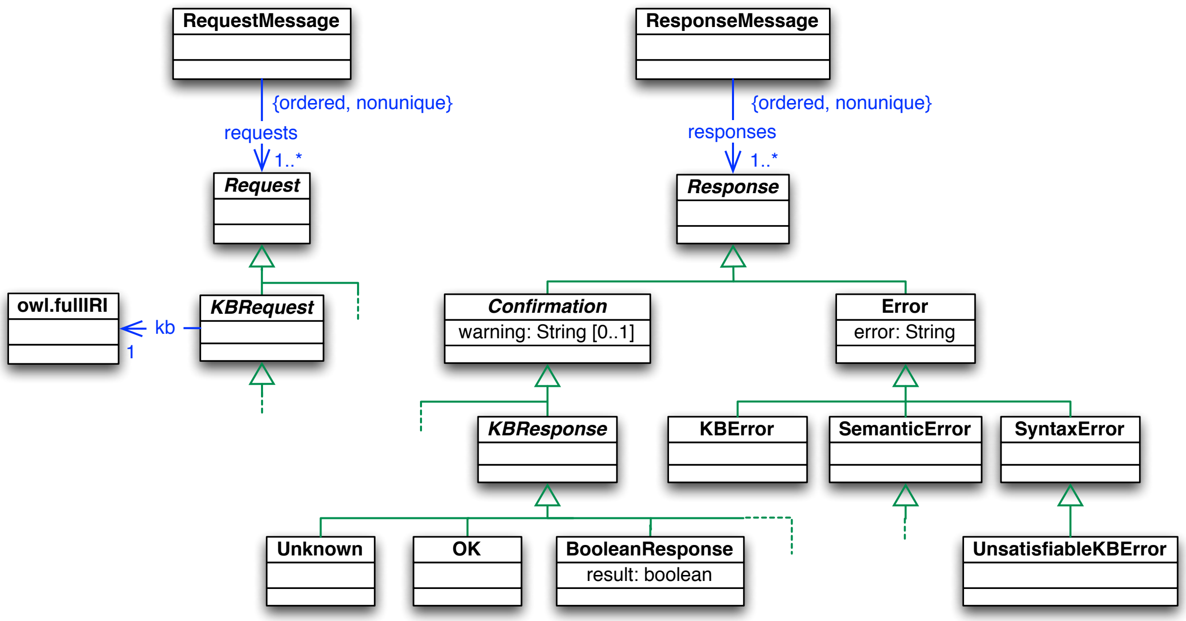

Fig. 1. Basic protocol objects

2.2. *Servers and Knowledge Bases*

The `GetDescription` request has to be supported by any OWLlink server, to allow clients to discover their identity and introspect their capabilities. The response to this request is a `Description`, providing information about the server's current state, including its name, its version, an optional identification message, the protocol version, the currently managed KBs, the supported extensions, and a set of configurations.

A `Configuration` is either a `Property` or a `Setting`. While properties are read-only, settings can be adjusted per KB at any time via a `Set` request. The settings given in a description indicate the server's defaults that hold for newly created KBs. The actual settings of a KB can be retrieved via `GetSettings`. While OWLlink defines the general format of configurations, it does not provide specific details on available configurations – these will be defined on a per-server basis. However, some configurations such as the language profile (namely `selectedProfile`, which lists the supported language fragments[2] as its type and the active profile as value) or the underlying semantic (`appliedSemantics`) of OWL 2 have to be supported by any OWLlink server. A server is free to refuse the adjustment of a setting at any point in the lifetime of a KB.

OWLlink servers can manage more than one KB simultaneously. A new KB is allocated within the OWLlink server by sending a `CreateKB` request. If the optional argument `kb` is given, the new KB is allocated with the given IRI, otherwise a fresh, server-generated IRI is used. On successful creation, a response named `KB` with the identifying IRI of the allocated KB is returned. An optional argument `name` allows to associate a name with a KB, which is then published to other clients (together with its IRI) within the server description. After disposal of a KB with the `ReleaseKB` request the server has to respond with an `KBError` to any requests to the respective KB IRI.

2.3. *Assertions*

OWLlink relies on the language primitives of OWL 2. Axioms can be added to a KB by a `Tell` request that refers to the various term constructors about classes, properties or facts defined in the OWL 2 specification [12]. A `Tell` request contains a (non-empty) set of OWL 2 axioms and will be answered with an `OK` response when successfully processed by the server (see Figure 2). In analogy to the definition of an ontology in OWL 2, an OWLlink KB is the cumulative set of all asserted axioms without duplicates with respect

[2]Can be any well-defined DL fragment (e.g. 'ALC') or OWL 1 and OWL 2 sublanguage (e.g. 'OWL 2 RL').

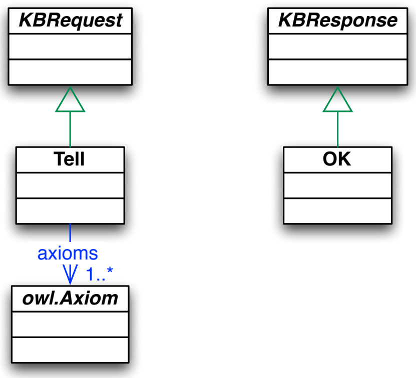

Fig. 2. Asserting axioms

to the structural equivalence of OWL 2. However, an OWLlink KB has neither an ontology IRI nor a version IRI, but is identified by a KB IRI.

In addition to sending axioms directly via successive `Tell` requests, OWLlink allows to read ontology documents via the `LoadOntologies` request. Such a request loads one or more ontologies, and optionally all imported ontologies, into a given KB by dereferencing the given IRIs. This request also offers an IRI mapping which allows to redirect ontology IRIs to physical ontology documents.

2.4. *Basic queries*

The OWLlink core includes a set of general requests for retrieving information about a KB. These so called basic asks cover common queries with respect to the given and inferred axioms. More complex queries are delegated to appropriate query extensions. To provide an informal overview, the table in the Appendix lists all of the basic asks with their response types and optional argument flags ("`neg`" for negative property assertions and "`dir`" for direct with respect to the subsumption relationship). Their semantics and a detailed description of their corresponding responses is given in the OWLlink structural specification [9]. The following briefly describes selected queries to show the key aspects of the interface.

One important query for an OWL 2 compliant reasoning system is to check whether a KB entails an axiom. The corresponding OWLlink `IsEntailed` request returns either `Unknown` or a `BooleanResponse` carrying the value true in case the supplied OWL 2 axiom is entailed by the respective KB with respect to the underlying semantics and false otherwise. To also support commonly used entailments such as the direct subsumers of a class or the direct types of an individual OWLlink also provides the `IsEntailedDirect` request. The latter is only applicable to a subset of the OWL 2 axioms (that is the sub-entity and class assertion axioms).

OWLlink responses contain structures that group equivalent entities. As an example, whenever a query returns a collection of classes, properties, or individuals whose elements potentially are equivalent to each other, the response is a set consisting of synonym sets (so called synsets). The OWLlink naming schema provides general rules on naming request and response identifiers including when and how to use prefixes such as "`Is`", "`Get`", or "`SetOf`". According to this naming schema, in case of class entities a synset is denoted by a `ClassSynset` element. For instance, the `GetTypes` request retrieves all classes an individual is an instance of. The corresponding response therefore returns one or more `ClassSynset` elements which contain at least one `owl.Class` (see Figure 3). Since this grouping might be

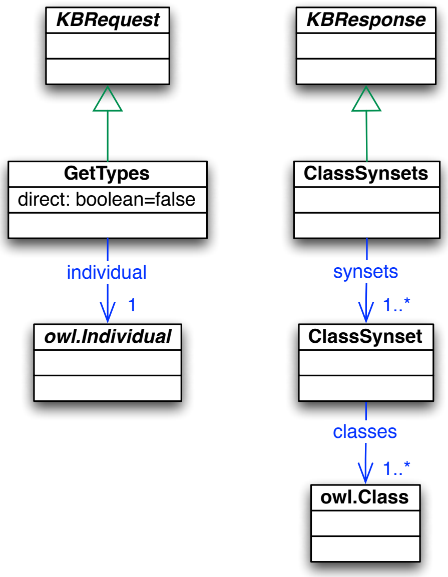

Fig. 3. Individual types request and response

an undesired overhead under some conditions (e. g. in case of a unique name assumption or global disjointness of classes), there are flattened versions for particular queries. The `GetFlattenedTypes` query, for instance, rules out any information about class equivalence from the response and simply returns a set of classes.

Another commonly used query refers to the class hierarchy of an ontology. The OWLlink `GetSubClassHierarchy` request retrieves the partial or complete class hierarchy of a KB. Since OWLlink aims

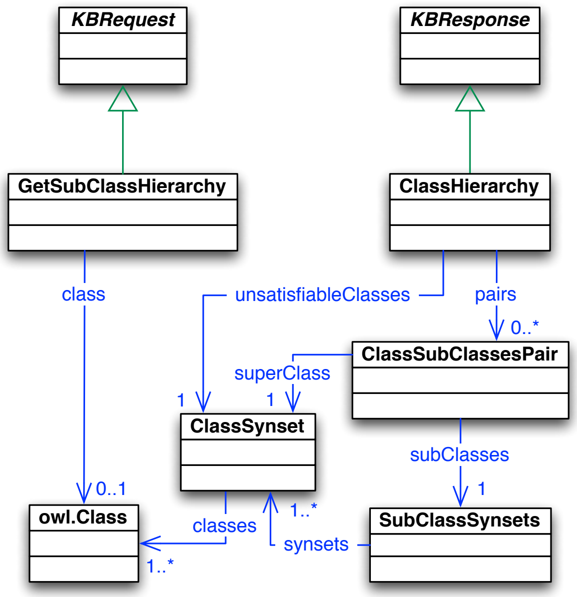

Fig. 4. Class hierachy request and response

at minimizing the size of its messages this response only returns the direct subsumption relationships and eliminates all those relationships, which refer to `owl:Nothing`. More precisely, it omits all those pairs of direct super-/subclasses, which contain the `owl:Nothing` class. The latter as well as all other unsatisfiable classes are carried as one extra synset within each `ClassHierarchy` response as depicted in Figure 4. With the help of this information an application can reconstruct the implicitly given but omitted relationships from any `ClassHierarchy` w.r.t. . `owl:Nothing`

Note that OWLlink offers requests for classifying or realizing a KB but does not require any explicit processing request before querying. The server has to take care about computing inferences and decide when it is appropriate to (re-)compute the internal data structures.

2.5. *Extensions*

The OWLlink core is extendable in terms of the supported language fragment, the offered services, as well as provided management tasks. An extension consists of a set of documents specifying the additional messages, a structural specification providing sufficient information about their meaning, and a document per supported binding defining the extra syntax. An extension should not redefine core OWLlink statements and has to adopt the given naming schema. A server reports the set of extensions supported in the `Description` object by listing their associated IRIs.

To date there are a couple of extensions under development, such as the *Told Data Access*, providing access to previously told axioms, and the *Ontology Based Data Access*, supporting the access to data stored in heterogeneous sources through a semantic layer. While the first extension allows to differentiate inferred from asserted knowledge, the second one can be used to maintain mappings between data in sources and KB entities.

A further extension, *Retraction* [10], enables the retraction of previously told axioms to efficiently update KBs without making the detour of a KB release and resubmission cycle. More precisely, the `Retract` request is the inverse of a `Tell` and removes a set of OWL 2 axioms from the given KB. The removal must be sensitive to the rules of structural equivalence of OWL 2. If all axioms of a retraction request are successfully removed from the KB, the server issues an `OK` response.

3. Bindings

An OWLlink binding defines a syntax for the requests and responses of the structural specification as well as a concrete transport protocol in order to practically enable communication among system components. This section introduces three such bindings namely XML expressions over HTTP, Functional expressions over HTTP, and S-Expressions over HTTP.

All transport related communication issues such as authentication, encryption, or compression need to be handled by the binding. Compression, for instance, often a key feature to decrease response latency in a remote scenario, is transparently supported by transport protocols such as HTTP.

3.1. *HTTP/XML binding*

HTTP/XML[14] is the default binding for OWLlink. It utilizes HTTP for exchanging requests an responses serialized as XML data between a reasoner and a client. OWLlink sessions are mapped to HTTP connections which are established upon sending the first request. More precisely, request messages are sent using HTTP Post requests. The actual content has to be valid with respect to the OWLlink XML Schema which has been obtained from the structural specification by a straightforward translation. This schema relies on the OWL 2 XML serialization [11] for the ontology primitives such that developers can employ existing parsers to read the

```
<RequestMessage
  xmlns="http://www.owllink.org/owllink#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:ret=
     "http://www.owllink.org/ex/retraction#">
 <CreateKB kb="http://family">
  <Prefix name="family"
    fullIRI="http://example.com/family/"/>
  <Prefix name="otherOnt"
    fullIRI="http://example.com/otherOnt/"/>
 </CreateKB>
 <LoadOntologies kb="http://family">
  <OntologyIRI
    IRI="http://owllink.org/primer.owl"/>
 </LoadOntologies>
 <GetTypes kb="http://family" direct="true">
  <owl:NamedIndividual
    abbreviatedIRI="family:John"/>
 </GetTypes>
 <Tell kb="http://family">
  <owl:SubClassOf>
   <owl:Class
     abbreviatedIRI="family:HappyPerson"/>
   <owl:Class
     IRI="http://example.com/family/Person"/>
  </owl:SubClassOf>
 </Tell>
 <GetObjectPropertySources kb="http://family">
  <owl:ObjectProperty
    abbreviatedIRI="family:hasWife"/>
  <owl:NamedIndividual
    abbreviatedIRI="family:Mary"/>
 </GetObjectPropertySources>
 <ret:Retract kb="http://family">
  <owl:SubClassOf>
   <owl:Class
     abbreviatedIRI="family:HappyPerson"/>
   <owl:Class
     abbreviatedIRI="family:Person"/>
  </owl:SubClassOf>
 </ret:Retract>
 <ReleaseKB kb="http://family"/>
</RequestMessage>
```

Fig. 5. Example `RequestMessage` in XML syntax

OWL 2 content, for example, within `Tell` and `Ask` requests. Figure 5 shows a sample request message that provides an impression of the XML serialization related to this binding. A corresponding server response is given in Figure 6.

The root element of any XML based binding has to be either a `RequestMessage` or a `ResponseMessage`. In correspondence of the notion of prefixes in OWL 2 ontologies, OWLlink also has an optional mechanism for declaring prefixes for KBs. However, in contrast to OWL 2, these prefixes are valid within the whole

```
<ResponseMessage
  xmlns="http://www.owllink.org/owllink#"
  xmlns:owl="http://www.w3.org/2002/07/owl#">
 <KB kb="http://family"/>
 <OK/>
 <ClassSynsets>
  <ClassSynset>
   <owl:Class
    abbreviatedIRI="family:Father"/>
  </ClassSynset>
  <ClassSynset>
   <owl:Class
    abbreviatedIRI="family:MyBirthdayGuests"/>
  </ClassSynset>
 </ClassSynsets>
 <OK/>
 <SetOfIndividualSynsets>
  <IndividualSynset>
   <owl:NamedIndividual
    abbreviatedIRI="family:John"/>
   <owl:NamedIndividual
    abbreviatedIRI="otherOnt:JohnBrown"/>
  </IndividualSynset>
 </SetOfIndividualSynsets>
 <OK/>
 <OK/>
</ResponseMessage>
```

Fig. 6. Example `ResponseMessage` in XML syntax

scope of the KB (successive assertions, queries, etc.) and not only within one particular ontology document. The OWLlink prefix declarations need to be defined at the time when creating the KB within the `CreateKB` request. Note that an OWLlink reasoner may bypass the serialization syntax of a binding by using the `Load-Ontologies` request which delegates the processing of loading ontologies to the server. This allows to utilize alternative parsers for formats other than XML such as OBO for example.

3.2. *HTTP/Functional binding*

The HTTP/Functional binding [20] also utilizes HTTP Post requests to transfer messages. However, the content is not encoded in XML. The OWLlink Functional binding employs the functional syntax of OWL 2 [12] to represent OWL 2 axioms and a closely related syntax for the OWLlink constructs. This allows to reuse OWL 2 functional parsers to implement this binding. The example request and response messages of Figure 7 and Figure 8 are equivalent to their XML counterparts given in the previous section, but utilize functional syntax.

```
NamespacePrefix("ret"
  <http://www.owllink.org/ext/retraction#>)
RequestMessage(
  CreateKB(Attribute(kb <http://family>)
    Prefix(Attribute(name "family")
      Attribute(fullIRI
        <http://example.com/family/>))
    Prefix(Attribute(name "otherOnt")
      Attribute(fullIRI
        <http://example.com/otherOnt/>)))
  LoadOntologies(kb <http://family>)
    OntologyIRI(
      Attribute(IRI
        <http://owllink.org/primer.owl>)))
  GetTypes(Attribute(kb <http://family>)
    Attribute(direct "true")
    family:John)
  Tell(Attribute(kb <http://family>)
    SubClassOf(family:HappyPerson
               family:Person))
  GetObjectPropertySources(
    Attribute(kb <http://family>)
    family:hasWife
    family:Mary)
  ret.Retract(Attribute(kb <http://family>)
     SubClassOf(family:HappyPerson
                family:Person))
  ReleaseKB(Attribute(kb <http://family>)))
```

Fig. 7. Example `RequestMessage` in Functional syntax

```
ResponseMessage(
  KB(Attribute(kb <http://family>))
  OK()
  ClassSynsets(
    ClassSynset(family:Father)
    ClassSynset(family:MyBirthdayGuests))
  OK()
  SetOfIndividualSynsets(
    IndividualSynset(family:John
                     otherOnt:JohnBrown))
  OK()
  OK())
```

Fig. 8. Example `ResponseMessage` in Functional syntax

Attribute values of OWLlink messages are encoded as `Attribute(<attribute-name> <attribute-value>)` terms. This schema shares the advantages of associative argument-passing methods (over position-based argument passing methods) and somehow resembles XML attributes.

Namespace prefix declarations allow to declare abbreviations for IRIs of OWLlink constructs. For example, the `NamespacePrefix` statement in Figure 7 allows to abbreviate `<http://www.owllink.org/ext/retraction#Retract>` as `ret.Retract` . The extent of namespace prefix declarations is the current OWLlink message. Moreover, the standard OWL 2 prefixes (`owl`, `rdf`, `rdfs`, `xsd`), as well as `ol` for the OWLlink namespace, can always be used without explicit declaration. Elements from these namespaces do not have to be qualified. Consequently, `SubClassOf` is a synonym for the qualified element `owl.SubClassOf`. This allows to transmit OWL 2 ontologies by simply embedding them in a surrounding `Tell` message. Namespace disambiguation (as for `owl.Literal` vs. `ol.Literal`) is resolved by taking the term context into account.

```
(NamespacePrefix () ret
  |http://www.owllink.org/ext/retraction#|)
(RequestMessage ()
  (CreateKB (:kb |http://family|)
    (Prefix (:name "family" :fullIRI
             |http://example.com/family/|))
    (Prefix (:name "otherOnt" :fullIRI
             |http://example.com/otherOnt/|)))
  (LoadOntologies (:kb |http://family|)
    (OntologyIRI
      (:IRI |http://owllink.org/primer.owl|)))
  (GetTypes (:kb |http://family|
             :direct "true")
    |family:John|)
  (Tell (:kb |http://family|)
    (SubClassOf
      |family:HappyPerson|
      |family:Person|))
  (GetObjectPropertySources
      (:kb |http://family|)
    |family:hasWife|
    |family:Mary|)
  (ret.Retract (:kb "http://family")
    (SubClassOf
      |family:HappyPerson|
      |family:Person|))
  (ReleaseKB (:kb |http://family|)))
```

Fig. 9. Example `RequestMessage` in S-Expression syntax

Adopting the spirit of the OWL 2 functional syntax, entities are referenced by pure IRIs, not boxed within additional entity constructions as in the OWLlink XML syntax. For example, the named individual representing Mary is encoded as just `family:Mary` instead of the boxed variant `NamedIndividual(family:Mary)`.

3.3. HTTP/S-Expression binding

The OWLlink HTTP/S-Expression binding [19] is tailored towards easy processability by OWLlink components utilizing a functional programming language such as Common Lisp. As the previous bindings it uses

```
(ResponseMessage ()
  (KB (:kb |http://family|))
  (OK ())
  (ClassSynsets ()
   (ClassSynset () |family:Father|)
   (ClassSynset () |family:MyBirthdayGuests|))
  (OK ())
  (SetOfIndividualSynsets ()
   (IndividualSynset ()
     |family:John|
     |otherOnt:JohnBrown|))
  (OK ())
  (OK ()))
```

Fig. 10. Example `ResponseMessage` in S-Expression syntax

HTTP for exchanging requests and responses. The content, however, is encoded in the OWLlink S-Expression syntax. This syntax can also be considered as a substitute for the KRSS language. The previous example request and response messages are rendered as S-Expressions in Figure 9 and Figure 10, respectively.

The OWLlink S-Expression syntax is defined in terms of a translation from the structural OWLlink specification. Consequently, this translation also specifies an S-Expression syntax for OWL 2. The `NamespacePrefix` mechanism is used in the same way as in the OWLlink Functional syntax to compensate for the lack of XML namespaces. Also the rules regarding unqualified OWL 2 elements are inherited from the Functional syntax.

The guiding design criterion for the S-Expression binding is *Lisp-readability*. Messages have to be parsable by the (case-preserving and unicode-aware) Common Lisp function `read` without requiring additional frameworks or machinery.

Since the OWL 2 functional syntax for (abbreviated) IRIs is unreadable by Lisp, the S-Expression syntax primarily uses Common Lisp symbols. Strings are acceptable as well if *not* used at operator (functor) position. As in Common Lisp, a symbol name must be surrounded by bars (`|...|`) in case it contains a colon, sharp (`#`), or some other non-alphanumeric characters (with the exception of the period character).

Attribute values are specified the Lisp-way, using keyword value pairs, with keywords prefixed by a colon. Even if an OWLlink message does not require attributes, an empty attribute list has to be used for reasons of uniformness and ease of implementation. This allows for the specification of additional (i. e., system-specific or future) attribute values without breaking the existing message syntax. In contrast, the always empty attribute list of OWL 2 terms is always omitted.

Since the OWL 2 functional syntax for `owl.typedLiteral` elements is unreadable by Lisp, a different syntax is used here. For example, the S-Expression for `"John Doe"^^xsd:string` is `(OWLLiteral "John Doe" "xsd:string")`.

To make OWLlink message more Lisp-like, some further provisions were made, e. g., boolean attribute values can be specified as strings `"true"` or `"false"`, but also as standard boolean values `t, nil`. More importantly, the rendering and hence verbosity of `Response` messages can be influenced via a set of *rendering options*. More details can be found in the binding specification [19].

4. Implementations

A number of implementations of OWLlink are already available. Besides applications that connect to reasoning engines acting as OWLlink servers, there are frameworks available that facilitate the incorporation of OWLlink support into existing and future systems.

The RacerPro[3] 2.0 reasoner implements the complete OWLlink protocol and its retraction extension. It not only supports the standard OWLlink HTTP/XML binding, but also the HTTP/Functional as well as the HTTP/S-Expression binding. Moreover, RacerPro can be used as converter between these different syntaxes. Its Lisp-based OWLlink module, including parsers and renderers for OWL 2, has been released as part of the open-source ontology framework OntoLisp.[4]

The OWLlink API[5] [13] realizes a client and server API for the Java platform (see Figure 11). The client adapter enables Java applications to access remote OWLlink servers. It builds on the widely used OWL API [7] and therefore enables OWLlink support for a large number of applications. The OWLlink API provides an extensible API that supports the full OWLlink core as well as the retraction extension. Additionally, it fully integrates with the OWL API structures by implementing its `OWLReasoner` interface. The server adapter, on the other hand, turns existing OWL API version 2 and 3 reasoners, such as FaCT++,[6] Pellet[7] or HermiT,[8] into OWLlink servers, ready for remote ac-

[3]http://www.racer-systems.com/
[4]http://ontolisp.sourceforge.net/
[5]http://owllink-owlapi.sourceforge.net/
[6]http://owl.man.ac.uk/factplusplus/
[7]http://clarkparsia.com/pellet/
[8]http://hermit-reasoner.com/

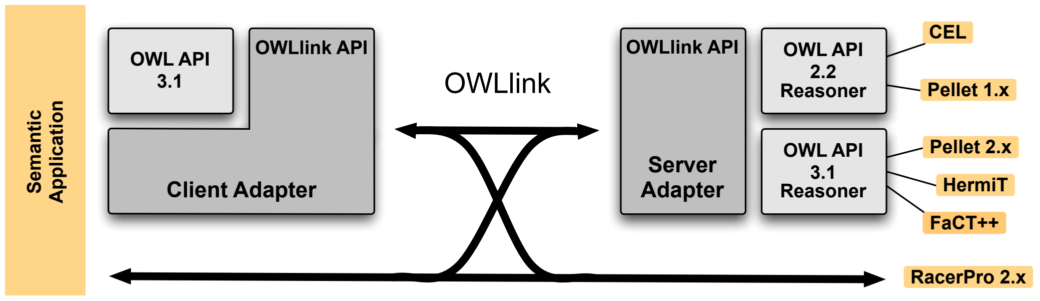

Fig. 11. OWLlink API architecture

cess. In addition, it provides a framework for reasoner developers to enhance existing reasoners with OWLlink functionality directly. Aside from the Java platform, OWLlink support is also available via the Thea library [18] for Prolog.

The implementation and adaption of OWLlink has also been started on the application side. The widely used ontology editor Protégé[9] version 4.1 has been extended to access OWLlink servers via a plug-in[5] that builds on the OWLlink API. The context-aware IYOUIT system [3] connects to RacerPro via its native OWLlink interface. Other applications, like the LarKC platform [8], still make use of the outdated DIG protocol, or utilize some proprietary protocol, like HERAKLES [2]. However, the implementors of those systems intent to replace their reasoner link-up by the common OWLlink protocol in the near future.

5. Outlook

As a W3C member submission, OWLlink is officially suggested to be contributed to any future W3C activity involved in the development of OWL or consider this as a starting point for work in a new working group. Even if there is no official activity on the radar right now, OWLlink can – and is intended to – be refined at least by its extension mechanism. As far as we see there is strong demand for extensions which provide a conjunctive query language as well as support for an OWL compatible rule language such as SWRL. As regards tool support and implemenations we see a solid starting point for adoption by the OWLlink API which basically allows to transform almost all reasoning systems with OWL API support into an OWLlink server.

[9]`http://protege.stanford.edu/`

References

[1] S. Bechhofer, R. Möller, and P. Crowther. The DIG Description Logic interface. In *Proc. of the Int. Workshop on Description Logics (DL'03)*, June 2003.
[2] J. Bock, T. Tserendorj, Y. Xu, J. Wissmann, and S. Grimm. A reasoning broker framework for OWL. In Hoekstra and Patel-Schneider [6].
[3] S. Böhm, J. Koolwaaij, M. Luther, and et al. Introducing IYOUIT. In *Proc. of ISWC'08*, volume 5318 of *LNCS*, pages 804–817. Springer Verlag, October 2008.
[4] B. Bulka and E. Sirin. PelletServer: HTTP & OWL 2 reasoning. In Sirin [16].
[5] S. Harris and A. Seaborne. SPARQL 1.1 Query Language. W3C Working Draft, World Wide Web Consortium, June 2010.
[6] R. Hoekstra and P. F. Patel-Schneider, editors. *Proc. of the OWLED'09 Workshop*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, October 2009.
[7] M. Horridge and S. Bechhofer. The OWL API. In Hoekstra and Patel-Schneider [6].
[8] Z. Huang. Initial evaluation and revision of plug-ins deployed in use-cases. EU IST FP7 Project Deliverable 4.7.1, The Large Knowledge Collider (LarKC), September 2009.
[9] T. Liebig, M. Luther, and O. Noppens. OWLlink: Structural Specification. Member Submission, World Wide Web Consortium, July 2010. `http://www.w3.org/Submission/owllink-structural-specification/`.
[10] T. Liebig and O. Noppens. OWLlink Extension: Retraction. Member Submission, World Wide Web Consortium, July 2010. `http://www.w3.org/Submission/owllink-extension-retraction/`.
[11] B. Motik, B. Parsia, and P. F. Patel-Schneider. OWL 2 Web Ontology Language: XML Serialization. W3C Recommendation, World Wide Web Consortium, October 2009.
[12] B. Motik, P. F. Patel-Schneider, and B. Parsia. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. W3C Recommendation, World Wide Web Consortium, October 2009.
[13] O. Noppens, M. Luther, and T. Liebig. The OWLlink API: Teaching OWL components a common protocol. In Sirin [16].
[14] O. Noppens, M. Luther, T. Liebig, and M. Wessel. OWLlink: HTTP/XML Binding. Member Submission, World Wide Web Consortium, July 2010. `http://www.w3.org/Submission/owllink-httpxml-binding/`.
[15] P. F. Patel-Schneider and B. Swartout. Description-Logic knowledge representation system specification from the KRSS group.

Working version (draft), 1993.

[16] E. Sirin, editor. *Proc. of the OWLED'10 Workshop*, CEUR Workshop Proceedings. CEUR-WS.org, 2010.

[17] M. Smith, I. Horrocks, M. Krötsch, and B. Glimm. OWL 2 Web Ontology Language: Conformance. W3C Recommendation, World Wide Web Consortium, October 2009.

[18] V. Vassiliadis, J. Wielemaker, and C. Mungall. Processing OWL 2 ontologies using Thea: an application of logic programming. In Hoekstra and Patel-Schneider [6].

[19] M. Wessel. OWLlink: HTTP/S-Expression Binding. Member Submission, World Wide Web Consortium, July 2010.

[20] M. Wessel and M. Luther. OWLlink: HTTP/Functional Binding. Member Submission, World Wide Web Consortium, July 2010. `http://www.w3.org/Submission/owllink-httpfunct-binding/`.

Appendix: List of Basic Asks and Responses

	Ask		KBResponse
KB Entities	GetAllClasses		SetOfClasses
	GetAllObjectProperties		SetOfObjectProperties
	GetAllDataProperties		SetOfDataProperties
	GetAllAnnotationProperties		SetOfAnnotationProperties
	GetAllIndividuals		SetOfIndividuals
	GetAllDatatypes		SetOfDatatypes
Status	IsKBSatisfiable		BooleanResponse
	IsKBConsistentlyDeclared		BooleanResponse
	GetKBLanguage		StringResponse
Schema	IsEntailed		BooleanResponse
	IsEntailedDirect		BooleanResponse
	IsClassSatisfiable		BooleanResponse
	GetSubClasses	*[dir]*	SetOfClassSynsets
	GetSuperClasses	*[dir]*	SetOfClassSynsets
	GetDisjointClasses		ClassSynsets
	GetEquivalentClasses		SetOfClasses
	GetSubClassHierarchy		ClassHierarchy
	IsObjectPropertySatisfiable		BooleanResponse
	IsDataPropertySatisfiable		BooleanResponse
	GetSubObjectProperties	*[dir]*	SetOfObjectPropertySynsets
	GetSubDataProperties	*[dir]*	SetOfDataPropertySynsets
	GetSuperObjectProperties	*[dir]*	SetOfObjectPropertySynsets
	GetSuperDataProperties	*[dir]*	SetOfDataPropertySynsets
	GetDisjointObjectProperties		ObjectPropertySynsets
	GetDisjointDataProperties		DataPropertySynsets
	GetEquivalentObjectProperties		SetOfObjectProperties
	GetEquivalentDataProperties		DataProperties
	GetSubObjectPropertyHierarchy		ObjectPropertyHierarchy
	GetSubDataPropertyHierarchy		DataPropertyHierarchy
Facts	GetTypes	*[dir]*	ClassSynsets
	GetFlattenedTypes	*[dir]*	Classes
	GetDifferentIndividuals		SetOfIndividualSynsets
	GetFlattenedDifferentIndividuals		SetOfIndividuals
	GetSameIndividuals		IndividualSynonyms
	GetObjectPropertiesOfSource	*[neg]*	SetOfObjectPropertySynsets
	GetDataPropertiesOfSource	*[neg]*	SetOfDataPropertySynsets
	GetObjectPropertiesOfTarget	*[neg]*	SetOfObjectPropertySynsets
	GetDataPropertiesOfLiteral	*[neg]*	SetOfDataPropertySynsets
	GetObjectPropertiesBetween	*[neg]*	SetOfObjectPropertySynsets
	GetDataPropertiesBetween	*[neg]*	SetOfDataPropertySynsets
	GetInstances	*[dir]*	SetOfIndividualSynsets
	GetObjectPropertyTargets	*[neg]*	SetOfIndividualSynsets
	GetDataPropertyTargets	*[neg]*	SetOfLiterals
	GetObjectPropertySources	*[neg]*	SetOfIndividualSynsets
	GetDataPropertySources	*[neg]*	SetOfIndividualSynsets
	GetFlattenedInstances	*[neg]*	SetOfIndividuals
	GetFlattenedObjectPropertyTargets	*[neg]*	SetOfIndividuals
	GetFlattenedObjectPropertySources	*[neg]*	SetOfIndividuals
	GetFlattenedDataPropertySources	*[neg]*	SetOfIndividuals