OWLlink: DIG for OWL 2

Thorsten Liebig[†], Marko Luther[‡], Olaf Noppens[†], Mariano Rodriguez[‡‡], Diego Calvanese[‡‡], Michael Wessel[⋆], Ralf Möller[°], Matthew Horridge[*], Sean Bechhofer[*], Dmitry Tsarkov[*], Evren Sirin[⊛]

[†]University of Ulm (Germany), [‡]DOCOMO Euro-Labs (Germany), [‡‡]Free University of Bozen-Bolzano (Italy), [⋆]Racer Systems (Germany), [°]Hamburg University of Technology (Germany), [*]University of Manchester (UK), [⊛]Clark & Parsia (USA)

Abstract. The OWLlink interface provides an implementation-neutral mechanism for accessing OWL reasoner functionality. In contrast to its DL-oriented predecessor DIG, OWLlink relies on OWL 2 for the primitives of the modelling language, and is thus fully compatible with the forthcoming incarnation of OWL. The OWLlink core introduced in this document covers (i) basic reasoner management, (ii) assertion of axioms and (iii) elementary ask functionality. The OWLlink extension mechanism allows to easily add any required functionality in a controlled way to the core language. We introduce OWLlink by providing a structural specification and a concrete binding of the interface that defines how OWLlink messages can be encoded in XML and sent over HTTP.

1 Introduction

OWLlink provides a declarative interface to OWL reasoners. It is intended to be a lightweight mechanism giving client applications access to reasoning functionality provided by a server. The OWLlink core defines primitives for handling and manipulation of OWL Knowledge Bases (KBs), such as asserting axioms, as well as primitives for asking questions about KB entities.

Early attempts to standardize an interface for Description Logic (DL) systems go back to the late 90s. At that time, most DL systems (e. g., FaCT [1] and RACER [2]) provided proprietary interfaces using a Lisp-like syntax more or less compliant with the KRSS syntax [3]. In 1999, Bechhofer et al. [4] defined a CORBA-based interface to FaCT using XML syntax, enlarging the scope to more expressive DLs than covered by the KRSS specification. However, both interfaces are more or less system-oriented and were quoted as being a handicap for application developers.

The initial DIG 1.0 specification of 2002 [5] was designed by the DL Implementation Group (DIG), a self-selecting assembly of researchers and developers associated with implementations of DL systems. Its intention was to overcome the limitations of the CORBA-FaCT interface, by adding support for assertional knowledge (not supported by FaCT at that time), reasoner identification and concrete domains, among others. Furthermore, the goal was to align with actual DL language fragments, such as those underlying DAML+OIL, and to provide an HTTP based interface for the client-server communication. The shortly following

version DIG 1.1 of 2003 [6] was quickly adopted by numerous reasoning engines, ontology editors and other applications. Despite its success there remained several shortcomings, such as limitations in the supported language fragment, a lack of elementary queries, and no mechanism to extend the protocol [7]. Most of the identified issues were fixed in the draft specification of DIG 2.0 [8] by adding the requested features and lifting the concept language to OWL 2 [9]. Finally, DIG 2.0 has now been renamed into OWLlink to reflect the important shift from its DL roots to the XML-based cousin OWL and the lack of backwards compatibility to DIG.

Since syntax and semantic of the OWLlink language primitives are based on OWL 2, the effort to support the core protocol with respect to parsing is reduced. Consequently, OWLlink inherits all of its underlying language concepts such as punning[1] and the OWL 2 notion of structural equivalence. However, it does not support any parts of OWL 2 beyond the level of axioms. In particular, it does not support imports and the notion of an ontology. Client applications have to resolve any issues related to the handling of imports before passing a KB to an OWLlink server. Under this view, the OWLlink interface is by far more abstract in terms of implementation languages and internal representation models than frameworks that provide access to data structures representing OWL ontologies such as the Java specific OWL API [10]. Furthermore, the OWLlink specification does not address issues such as stateful connections, transactions, authentication, encryption, compression, concurrency, multiple clients and so on. Yet some features (such as authentication, encryption and compression) might be transparently provided by the access protocol (e. g., HTTP/1.1).

OWLlink is specified in two parts: the first part defines the abstract protocol, and the second part defines a binding of the protocol into a concrete transport mechanism. The abstract protocol is introduced in Sect. 2 and Sect. 3, its binding to HTTP/1.1 and XML Schema is given in Sect. 4. The full specification can be found online.[2]

2 Basic Protocol

OWLlink is a client-server protocol that makes a number of assumptions.

- The connection to the server is stateless and clients are not identified.
- OWLlink does not provide any support for modularity of Knowledge Bases.[3]
- There is no explicit classification request. The server will decide when it is appropriate to, for example, compute the classification hierarchy of concepts.

The following definitions use a very limited subset of UML class diagram notation, reusing many UML classes provided by the OWL 2 specification [9]. The

[1] Punning eliminates the need for introducing names. Each name can independently be used as the name of a property, a class, a datatype, or an individual.
[2] http://www.owllink.org
[3] A KB is a collection of facts and axioms.

names of abstract classes (that is, the classes that are not intended to be instantiated) are written in italic. The names of all OWL 2 UML classes are prefixed with ox. to emphasize that they are not defined in the OWLlink specification.

2.1 Sessions and Messages

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 mechanism used to access a OWLlink server.

OWLlink servers are allowed to service several clients concurrently; however, interaction within one session is not concurrent. A session is assumed to transport requests and responses sequentially ordered. Each request should be processed by the server such that the results are the same as if the requests were processed sequentially in the order they were dispatched.

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 separately. Therefore, OWLlink requests are bundled into messages. A `RequestMessage` object encapsulates a list of `Request` objects, whereas a `ResponseMessage` object encapsulates a list of `Response` objects (cf. Fig. 1).

2.2 Confirmation and Error Handling

Each request in OWLlink must be answered by a corresponding response. If a request has been processed successfully, the type of the response returned depends on the request and may contain additional data. If a request does not produce any specific data, the server should still return a `Confirmation` response (or subtype thereof) to the client. Any confirmation may carry a warning in terms of a string intended to be meaningful to a human user.

If a request fails for some reason, the server should return an `Error` response to the client containing a message specifying the cause for failure. Specific error classes are defined to report syntactic violations (`SyntaxError`), semantic problems (`SemanticError`) and issues regarding the management of a Knowledge Base (`KBError`). 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, after sending the `Error` response, the server should close the session.

2.3 Identification and Status

All OWLlink servers must support the `GetDescription` request, to allow clients to discover its identity and introspect its capabilities. The response to this request is a `Description`, providing information about the servers current state,

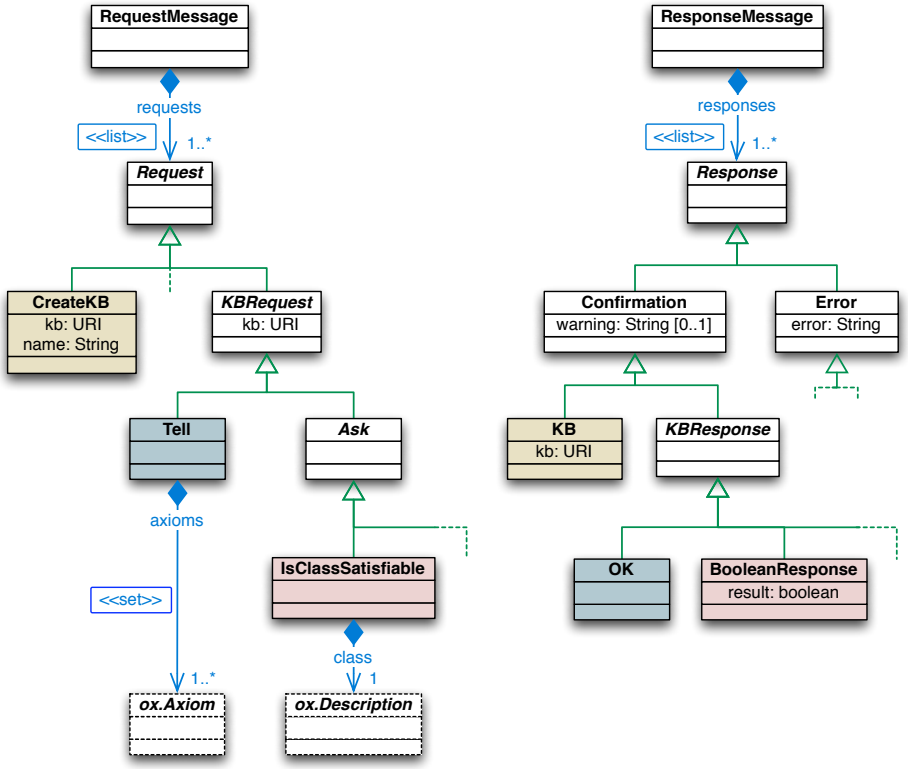

Fig. 1. Basic protocol objects

including: the name of the server, its version, an optional identification message, the protocol version, the currently managed public and named KBs (see Sect. 2.4), the supported extensions (see Sect. 2.5), 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 servers defaults that hold for newly created KBs. The actual settings 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, the following configurations have to be supported by any OWLlink server.

selectedProfile A configuration that names the default profile of the server in case of a `Description` response and the active profile in a `Settings` response. Its type restricts the possible values to the list of language fragments supported by the server. Any well-defined DL fragment might be referenced by name in this list, including, but not limited to, the ones defined by the OWL 2 Profiles [11] such as EL++, DL-Lite, OWL-R DL, etc.

ignoresAnnotations OWL 2 provides mechanisms for associating annotations with KB axioms and entities. If the management of annotations might incur unnecessary implementation and run-time overhead, this configuration allows to instruct an OWLlink server to ignore annotations.[4]

ignoresDeclarations If a server is instructed to ignore declarations via this configuration, all received declaration axioms are treated as not being send. In case a server deals with declarations a declaration axiom is treated as all other axioms (e.g. being sensible to structural equivalence).

uniqueNameAssumption A configuration specifying whether the server employs the Unique Name Assumption (UNA) such that different names always refer to different entities in the world.[5]

2.4 Knowledge Base Management

Servers may manage several KBs simultaneously. Therefore, each KB is identified with a unique URI on creation. These URIs provide a handle that identifies the particular incarnation of the KB in the particular server. Most requests are derived from the `KBRequest` class, which requires a `kb` argument identifying the KB to which the request applies (cf. Fig. 1). If a KB with the given URI cannot be found, the server should return a `KBError` object.

A new KB is allocated within the server by sending a `CreateKB` request. If the optional URI argument `kb` is given, the new KB is allocated with this URI, otherwise a fresh (server-generated) URI is used. However, if the given URI is already in use by another KB allocated before or the KB could not be allocated for some other reasons (e.g., a low memory condition), the response to the `CreateKB` request is a `KBError`. The optional `name` argument of a `CreateKB` request allows to associate a name with a KB on creation. Such KB names do not have to be unique and thus cannot be used to identify a certain KB. However, named KBs are published together with their identifying KB URI in the servers `Description` object to be accessible from multiple clients.

The use of unique URIs allows to sidestep some of the issues relating to multiple clients. If a client chooses not to share a KB URI with another client (either directly or by making it public through passing a name on creation), then the client can be sure that it is the only one interacting with that KB.[6]

Different clients of the same server, like KB browsers or explanation tools, however, may want to share KBs by sharing URIs or making KBs public through naming – it is then the clients' responsibility to manage and coordinate this sharing. The possibility to pass an identifying URI along with a `CreateKB` followed by a mixture of arbitrary tell and ask request allows OWLlink KBs to stand alone – a situation that is useful, say for test suites and benchmarking.

[4] A server configured this way should still accept annotations.

[5] OWL does not make the UNA [9]. Still, a client might choose to enable the UNA in case the reasoning performance is of utmost importance.

[6] This is not entirely the case, as there is a (very small) chance that a malicious client could guess a URI which is in use.

2.5 Extensions

Different reasoners support different language fragments and different reasoning facilities. To support these differences and future developments, OWLlink is extensible. Current candidates for extensions are:

Axiom Retraction Adds the ability to retract previously told axioms from KBs. Axiom retraction is sensitive to the rules of structural equivalence.[7]

Told Information Retrieval Provides access to previous told KB entities by returning requested portions of axioms – structurally exactly as given in previous tell messages.[8]

Ontology Based Data Access (OBDA) Extends OWLlink by the concepts of *data source* and *mapping* to support OBDA architectures facilitating the integration of data from existing data repositories.[9]

Other extensions are on the way such as a conjunctive query, a concrete domain interface, a non-standard inferences, and an explanation interface extension.

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 their syntax.

All documents must be available on the Web. A server reports the set of extensions supported for the binding used by a `GetDescription` request as part of the resulting `Description` object by listing their URIs without extension.

3 Tells & Asks

3.1 Tell Language

As mentioned before, OWLlink relies on the language primitives of OWL 2. With respect to the tell requests – those message parts which add axioms to a KB – this basically means that OWLlink refers to the various axioms about classes, properties or facts defined in Sect. 6 of the OWL 2 specification [9]. As depicted in Fig. 1 a `tell` request contains a set of one or more OWL 2 axioms and will be answered with a `OK` response when successfully processed by the server.

3.2 Basic Asks

The OWLlink core includes a set of general requests for retrieving information about KB entities as well as entailed facts. These so called basic asks only cover very common queries with respect to the given and inferred axioms of the KB. They substantially extend the query interface of DIG 1.1 with requests needed to be more aligned with OWL 2 or which have been missed by DIG user. In order to provide an informal overview the following table lists all of them. Their

[7] http://www.owllink.org/ext/retraction-20081001/
[8] http://www.owllink.org/ext/told-access-20081001/
[9] http://obda.inf.unibz.it/dig-11-obda/

semantic and a detailed description of their corresponding responses is given in the OWLlink structural specification.

	Ask	**KBResponse**
KB Entities	GetAllClasses	SetOfClasses
	GetAllObjectProperties	SetOfObjectProperties
	GetAllDataProperties	SetOfDataProperties
	GetAllAnnotationProperties	SetOfAnnotationProperties
	GetAllIndividuals	SetOfIndividuals
	GetAllDatatypes	SetOfDatatypes
Status	IsKBSatisfiable	BooleanResponse
	IsKBStructurallyConsistent	BooleanResponse
	GetKBLanguage	StringResponse
Schema	IsClassSatisfiable	BooleanResponse
	IsClassSubsumedBy	BooleanResponse
	AreClassesDisjoint	BooleanResponse
	AreClassesEquivalent	BooleanResponse
	GetSubClasses	SetOfClassSynsets
	GetSuperClasses	SetOfClassSynsets
	GetEquivalentClasses	SetOfClasses
	GetClassHierarchy	ClassHierarchy
	Are[O|D]PropertiesEquivalent	BooleanResponse
	Is[O|D]PropertySatisfiable	BooleanResponse
	Are[O|D]PropertiesDisjoint	BooleanResponse
	Is[O|D]PropertySubsumedBy	BooleanResponse
	Is[O|D]PropertyXXX	BooleanResponse
	GetSub[O|D]Properties	SetOf[O|D]PropertySynsets
	GetSuper[O|D]Properties	SetOf[O|D]PropertySynsets
	GetEquivalent[O|D]Properties	SetOf[O|D]Properties
	Get[O|D]PropertyHierarchy	[O|D]PropertyHierarchy
Facts	AreIndividualsEqual	BooleanResponse
	AreIndividualsDifferent	BooleanResponse
	IsInstanceOf	BooleanResponse
	GetTypes	SetOfClassSynsets
	GetEquivalentIndividuals	SetOfIndividuals
	Get[O|D]PropertyOfSource	SetOf[O|D]PropertySynsets
	GetObjectPropertyOfFiller	SetOfObjectPropertySynsets
	GetDataPropertyOfConstant	SetOfDataPropertySynsets
	Get[O|D]PropertiesBetween	SetOf[O|D]PropertySynsets
	GetInstances	SetOfIndividualSynsets
	GetObjectPropertyFillers	SetOfIndividualSynsets
	GetDataPropertyFillers	SetOfConstants
	Get[O|D]PropertySources	SetOfIndividualSynsets
	GetFlattenInstances	SetOfIndividuals
	GetFlattenObjectPropertyFillers	SetOfIndividuals
	GetFlattenObjectPropertySources	SetOfIndividuals

Within this table "[O|D]" abbreviates that this query exists in two flavors, either for Object- as well as for DataProperties. Furthermore the "XXX" in

`Is[O|D]PropertyXXX` is a wild-card for the various characteristics a property can have.

Responses of type `SetOfXXXSynset` consist of zero or more synsets of an OWL 2 entity such as Individual, Object- or DataProperty, or OWLClass. Such a synset is a set of one or more elements whose members are all equivalent to each other, i. e. for which mutual equivalence is entailed from the axioms of the KB. In case this information about equivalence sets is not needed (e. g. due to UNA) there are so called flattened variants for those asks which deal with return sets consisting of individuals.

4 HTTP/XML Binding

The HTTP/XML binding of OWLlink uses HTTP as its underlying communication protocol for exchanging XML content between a reasoner and a client. Other bindings may utilize SOAP or a particular Remote Message Invocation protocol. Technically, within the HTTP/XML environment, the reasoner has to accept HTTP POST request and responds as appropriate according to the OWLlink specification. In order to alleviate the verbose nature of any XML-based protocol OWLlink servers should support the use of the standard compression technology for Content-Encoding of the HTTP/1.1 transport mechanism.

4.1 Sessions and Messages

An OWLlink session is mapped to an HTTP connection and is typically established upon sending the first request. HTTP servers are in general allowed to close the HTTP connection at their own discretion, which in effect terminates the OWLlink session.[10] In order to prevent this from happening, the client can include the keepAlive in the header of each HTTP request it sends.

4.2 XML Schema

The XML schema is obtained by a straightforward translation of the objects from the structural specification: in general, the names of XML elements correspond to the names of the corresponding UML classes. It relies on the OWL 2 XML serialization for the primitives of the ontology language, and is therefore fully compatibly with OWL. As a result, implementors of the XML binding can re-use their implementation of OWL 2 parsers to read the OWL 2 specific contents of the tell and ask primitives.

Each OWLlink request (i. e., management, tell and ask primitives) must be acknowledged by a corresponding OWLlink response that are pooled within a `RequestMessage` respectively a `ResponseMessage`. Following a straightforward translation of the OWLlink specification into an XML schema the content body of a HTTP message is either the root element `RequestMessage` or a `ResponseMessage` containing requests resp. responses as can be seen in Fig. 2. Some more detailed examples are given in Appendix A.

[10] Note that closing an OWLlink session does not imply the release of any KB.

```
<owllink:RequestMessage>
    [request #1]
    [request #2]
    [ ... ]
    [request #n]
</owllink:RequestMessage>
```

```
<owllink:ResponseMessage>
    [response #1]
    [response #2]
    [ ... ]
    [response #n]
</owllink:ResponseMessage>
```

Fig. 2. `RequestMessage` and `ResponseMessage`

5 Discussion

Strong evidence suggests that OWL 2 will become an important standard for representing ontologies. According to the W3C its featured usage is to enable applications to meaningfully process information by means of reasoning. However, from the application's point of view this requires not only a language standard for ontologies but also a standard way to interact with components which supply reasoning or other services. OWLlink adds this latter piece of the ontology infrastructure by providing an extensible protocol for communication with OWL reasoning systems. The OWLlink protocol facilitates client applications to configure a reasoner, to transmit OWL 2 ontologies or fractions thereof, and to access reasoning services via a set of basic queries. Furthermore, OWLlink is flexible in that it allows to add any desired functionality by defining a corresponding extension.

The current OWLlink specification is still a draft and obviously cannot be finalized before the OWL 2 language specification has reached W3C recommendation status. In the meantime the OWLlink specification itself has to undergo a process where interface details (e. g. server descriptions or the basic asks) need to be reviewed by potential server and client developers as well as users. Therefore, anyone who feels addressed by this initiative is requested to join the discussion and to provide feedback to the OWLlink working group[11] as well as to the developers of those OWL components they would like access via OWLlink.

In fact, the developers of the established and most widely used reasoning systems RacerPro, Pellet and FaCT++ attentively follow the development of OWLlink and some already have announced to support the protocol. The developers of the QuOnto system are in the process of migrating their current OBDA extension for DIG 1.1 to the new OWLlink proposal (cf. Sect. 2.5). Various other developers of client applications, for example ontology editing and browsing tools such as the Protégé 4 or OntoTrack, also commited to adopt OWLlink. The same holds for the Java based OWL-API.

OWLlink provides a framework for accessing management and reasoning services around the forthcoming OWL 2. It is flexible to cope with future demands with respect to the underlying language as well as services. Beyond that it is

[11] There is a mailing list with public archive you can subscribe and post to. Please see https://imap.uni-ulm.de/lists/info/public-owllink-comments

designed on a structural level which enables many different bindings to concrete transport mechanisms. Besides the introduced XML/HTTP binding there are many other options. For instance there is a draft with respect to a binding aming at transporting OWL 2 functional prefix style syntax over HTTP (or a simple TCP socket connection).[12] The latter would provide a modern substitute for the outdated KRSS standard. Even an in-memory binding within one application, by mapping the OWLlink primitives to interfaces for instance, is a desired and sensible usage of OWLlink. In any case using OWLlink comes with the benefit of having a well-defined interface that allows to easily plug different OWL-aware components together.

References

1. Horrocks, I.: Using an expressive description logic: FaCT or fiction? In: Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98). (1998) 636–647
2. Haarslev, V., Möller, R.: Description of the RACER system and its applications. In: Proc. of the Int. Workshop on Description Logics. (2001)
3. Patel-Schneider, P.F., Swartout, B.: Description-Logic knowledge representation system specification from the KRSS group. Working version (draft) (1993)
4. Bechhofer, S., Horrocks, I., Patel-Schneider, P., Tessaris, S.: A proposal for a Description Logic interface. In: Intern. Workshop on Description Logics. (1999)
5. Bechhofer, S.: The DIG Description Logic interface: DIG/1.0. Technical report, University of Manchester (2002)
6. Bechhofer, S., Möller, R., Crowther, P.: The DIG Description Logic interface. In: Proc. of the Int. Workshop on Description Logics (DL'03). (2003)
7. Dickinson, I.: Implementation experience with the DIG 1.1 specification. Technical Report HPL-2004-85, Hewlett-Packard (2004)
8. Turhan, A.Y., Bechhofer, S., Kaplunova, A., Liebig, T., Luther, M., Möller, R., Noppens, O., Patel-Schneider, P., Suntisrivaraporn, B., Weithöner, T.: DIG2.0 – towards a flexible interface for Description Logic reasoners. In: Proc. of the OWL Experiences and Directions Workshop at the ISWC'06. (2006)
9. Motik, B., Patel-Schneider, P.F., Horrocks, I.: OWL 2 Web Ontology Language: Structural specification and functional-style syntax. W3C Working Draft, 11 April 2008, World Wide Web Consortium (2008)
10. Horridge, M., Bechhofer, S., Noppens, O.: The OWL API. In: Proc. of the 3rd OWL Experiences and Directions Workshop at the ESWC'07. (2007)
11. Grau, B.C., Motik, B., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language: Profiles. W3C Working Draft, 11 April 2008, World Wide Web Consortium (2008)

[12] http://www.owllink.org/owllink-httpsexpr-20081001/

A OWLlink XML Binding Examples

A sample configuration request which is typically send at the initialization phase to introspect the OWLlink server capabilities:

```
<RequestMessage xmlns="http://www.owllink.org/owllink-xml">
  <GetDescription/>
</RequestMessage>
```

A corresponding reasoner response with information about the offered OWLlink protocol version, reasoner version as well as offered options and settings followed by a list of supported extensions and public KBs:

```
<!DOCTYPE ResponseMessage [
 <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
 <!ENTITY ole "http://www.owllink.org/owllink-ext-xml/"> ]>
<ResponseMessage xmlns="http://www.owllink.org/owllink-xml"
    xmlns:ox="http://www.w3.org/ns/owl2-xml"
    xmlns:ol="http://www.owllink.org/owllink-xml">
    <Description ol:name="MyReasoner">
      <OWLlinkVersion ol:major="1" ol:minor="0"/>
      <ReasonerVersion ol:major="1" ol:minor="0"/>
      <Setting ol:key="selectedProfile">
        <List><OneOf>
                 <Literal ol:URI="&xsd;string">OWL-DL</Literal>
                 <Literal ol:URI="&xsd;string">OWL-R DL</Literal>
        </OneOf></List>
        <Literal>OWL-DL</Literal>
      </Setting>
      <Setting ol:key="uniqueNameAssumption">
        <Datatype ol:URI="&xsd;boolean"/>
        <Literal>true</Literal>
      </Setting>
      <Property ol:key="ignoresAnnotations">
        <Datatype ol:URI="&xsd;boolean"/>
        <Literal>true</Literal>
      </Property>
      <Property ol:key="ignoresDeclarations">
        <Datatype ol:URI="&xsd;boolean"/>
        <Literal>true</Literal>
      </Property>
      <SupportedExtension ol:URI="&ole;retraction-xml"/>
      <SupportedExtension ol:URI="&ole;told-xml"/>
      <PublicKB ol:kb="KB_2" ol:name="My KB 2"/>
    </Description>
</ResponseMessage>
```

The following is an example of a request (and its response) containing a "test-suite" or "benchmark-style" OWLlink message starting with the creation of a KB which is followed by a mixture of tells and asks:

```
<!DOCTYPE ResponseMessage [
 <!ENTITY owl "http://www.w3.org/2002/07/owl#"> ]>
<RequestMessage xmlns="http://www.owllink.org/owllink-xml"
                xmlns:ol="http://www.owllink.org/owllink-xml"
                xmlns:ox="http://www.w3.org/ns/owl2-xml#">
 <!-- KB management -->
 <CreateKB ol:kb="KB_1"/>
 <CreateKB ol:kb="KB_2"
           ol:name="My KB 2"/>
 <!-- Some tells in KB_1-->
 <Tell ol:kb="KB_1">
  <ox:SubClassOf>
    <ox:OWLClass ox:URI="B"/>
    <ox:OWLClass ox:URI="A"/>
  </ox:SubClassOf>
  <ox:SubClassOf>
    <ox:OWLClass ox:URI="C"/>
    <ox:OWLClass ox:URI="A"/>
  </ox:SubClassOf>
  <ox:EquivalentClasses>
    <ox:OWLClass ox:URI="D"/>
    <ox:OWLClass ox:URI="E"/>
  </ox:EquivalentClasses>
  <ox:ClassAssertion>
    <ox:OWLClass ox:URI="A"/>
    <ox:Individual ox:URI="iA"/>
  </ox:ClassAssertion>
 </Tell>
 <!-- Some asks -->
 <GetAllClasses ol:kb="KB_1"/>
 <GetEquivalentClasses ol:kb="KB_1">
    <ox:OWLClass ox:URI="D"/>
 </GetEquivalentClasses>
 <IsClassSubsumedBy ol:kb="KB_1">
   <ox:OWLClass ox:URI="&owl;Thing"/>
   <ox:OWLClass ox:URI="&owl;Nothing"/>
 </IsClassSubsumedBy>
 <GetSubClasses ol:kb="KB_1">
   <ox:OWLClass ox:URI="C"/>
 </GetSubClasses>
 <!--Some tells in another KB -->
 <Tell ol:kb="KB_2">
   <ox:SubClassOf>
     <ox:OWLClass ox:URI="A"/>
     <ox:OWLClass ox:URI="B"/>
   </ox:SubClassOf>
 </Tell>
 <!-- KB management -->
 <ReleaseKB ol:kb="KB_1"/>
 <!-- One more ask -->
 <GetAllClasses ol:kb="KB_1"/>
</RequestMessage>
```

```
<!DOCTYPE ResponseMessage [
 <!ENTITY owl "http://www.w3.org/2002/07/owl#"> ]>
<ResponseMessage xmlns="http://www.owllink.org/owllink-xml"
                 xmlns:ol="http://www.owllink.org/owllink-xml"
                 xmlns:ox="http://www.w3.org/ns/owl2-xml#">
 <!-- KB management -->
 <KB ol:kb="KB_1"/>
 <KB ol:kb="KB_2"/>
 <!-- tell -->
 <OK/>
 <!-- ask -->
 <SetOfClasses>
   <ox:OWLClass ox:URI="A"/>
   <ox:OWLClass ox:URI="B"/>
   <ox:OWLClass ox:URI="C"/>
   <ox:OWLClass ox:URI="D"/>
   <ox:OWLClass ox:URI="E"/>
 </SetOfClasses>
 <SetOfClasses>
  <ox:OWLClass ox:URI="E"/>
 </SetOfClasses>
 <BooleanResponse ol:result="false"/>
 <SetOfClassSynsets>
   <ClassSynset>
    <ox:OWLClass ox:URI="&owl;Nothing"/>
   </ClassSynset>
 </SetOfClassSynsets>
 <!--Some tells in another KB -->
 <OK/>
 <!-- KB management -->
 <OK/>
 <!-- One more ask -->
 <KBError errorMessage=
     "KB KB_1 does not exist!"/>
</ResponseMessage>
```