The DIG Description Logic Interface

Sean Bechhofer[†] Ralf Möller[‡]

Peter Crowther[§]

[†]Department of Computer Science, University of Manchester.

[‡]Univ. of Applied Sciences, Fachhochschule Wedel.

[§]Melandra Limited, UK.

Abstract

The Semantic Web has seen increased interest in the use of Description Logic technology to support ontologies, in particular, the use of classification and consistency checking. This requires that the functionality of DL reasoners be made readily available to applications. We present a simple interface definition for DL systems that facilitates their use in systems and describe its implementation in two DL systems.

1 Introduction

Ontologies have become an increasingly important research topic. This is a result both of their usefulness in a range of application domains [17, 14, 16], and of the pivotal rôle that they are set to play in the development of the *Semantic Web*

The Semantic Web vision, as articulated by Tim Berners-Lee [7], is of a Web in which resources are accessible not only to humans, but also to automated processes, e.g., automated "agents" roaming the web performing useful tasks such as improved search (in terms of precision) and resource discovery, information brokering and information filtering. The automation of tasks depends on elevating the status of the web from machine-readable to something we might call machine-understandable. The key idea is to have data on the web defined and linked in such a way that its meaning is explicitly interpretable by software processes rather than just being implicitly interpretable by humans.

Thus the provision of languages with well-defined semantics is seen as key to enabling the notion of machine-understandability, and the latest generation of Web Ontology languages such as OIL [9], DAML+OIL [1] and OWL [8] have placed much emphasis on such semantics. In addition, Description Logic languages [2] have emerged as a leading contender for the logical representation of ontologies. DLs are a family of languages that provide fragments of first-order logics that are restricted in such a way to allow tractable reasoning. DAML+OIL was effectively a description logic and the definition of OWL is being layered [8] in such a way as to provide subsets of the language that map to an expressive Description Logic. The expressivity of the OWL-Lite and OWL-DL layers has been tailored to facilitate tractable reasoning over ontologies represented in those languages.

1.1 DL Systems

In the past, description logic (DL) systems have presented the application programmer with a functional interface, often defined using a Lisp-like syntax. Such interfaces may be more or less complex, depending on the sophistication of the implemented system, and may be more or less compliant with a specification such as KRSS [15].

The Lisp style of the KRSS syntax reflects the fact that Lisp is still a common implementation language for DLs. This can create considerable barriers to the use of DL systems by application developers, who often prefer other languages (in particular the currently ubiquitous Java), and who are becoming more accustomed to component based software development environments. This is of increasing importance given current interest in Web Services and service based architectures. In addition, KRSS can be seen as more of a *language description* than an API.

In a distributed, web-based environment, a DL might naturally be viewed as a self contained component, with implementation details, and even the precise location at which its code is being executed, being hidden from the application [6]. This approach has several advantages: the issue of implementation language is finessed; the API can be defined in some standard formalism intended for the purpose; a mechanism is provided for applications to communicate with the DL system, either locally or remotely; and alternative DL components can be substituted without affecting the application.

This approach was adopted in the CORBA-FaCT system [4], where a CORBA interface was defined for a description logic reasoner. This wrapping of the FaCT reasoner facilitated the successful use of the reasoner in applications such as OilEd [3] and ICOM [10]. Although useful, CORBA-FaCT suffered from a number of inadequacies. The concept langage does not cover concrete domains, and as the interface was largely based on the FaCT reasoner (which did not support an A-Box at the time), functionality relating to A-Boxes was missing. In addition the concept identifiers allowed were closely tied to the underlying Lisp implementation (case insensitive strings of essentially alphanumeric characters). This introduces problems when trying to reason over languages such as DAML+OIL where classes are referred to using URIs[1].

The RACER [12] system adopted a slightly different mechanism, providing a socket based interface for use by client applications. This again provides a language neutral API, but the lower-level interface places more onus on the client programmer.

The Description Logic Implementation Group (DIG)[2] is a small self-selected group of DL system implementors. DIG was formed with the intention of sharing implementation experiences and moving towards standard system architectures for DL systems. One of the first activities undergone by DIG was the development of the interface as described in this paper.

2 The DIG interface

The DIG interface described here provides a basic API to a DL system, and should be considered as a *Level 0* specification – in its current version it contains just enough

[1] Of course such problems are not insurmountable, but do provide barriers to the ease of use of the systems.

[2] `http://dl.kr.org/dig`

functionality to enable tools such as OilEd [3] to use a DL reasoner. It does not provide what we might truly call a *reasoning service*, but rather helps to insulate applications from the location and implementation language of a DL reasoner. The specification does not address issues such as stateful connections, transactions, reasoner preferences and so on. There is nothing inherently new in this specification – it is effectively an XML Schema for a DL concept language along with ask/tell functionality. Along with the definition of the interface, however, there is a commitment from implementors of leading DL reasoners (such as FaCT [13], RACER [12] and Cerebra[3]) to provide implementations conforming to the specification (see Section 7). This will truly allow us to build plug and play applications where alternative reasoners can be seamlessly integrated into our systems. Applications need not know the details of the underlying reasoner being used at a particular time, and can instead access reasoning engines using a common interface.

The remainder of the paper provides a brief overview of the interface. Space precludes us from providing detailed descriptions of message formats, but further information (along the latest schemas) can be found at:

`http://dl-web.man.ac.uk/dig`

Assumptions A number of assumptions have been made for this initial specification.

- The specification is agnostic as to multiple client connections. Multi-threaded implementations of a reasoner may be provided, but no guarantees are made as to the semantics when clients attempt to simultaneously update and query.
- The connection to the reasoner is effectively stateless. Clients are not identified to the reasoner, thus the reasoner will not distinguish between clients and maintain any kind of consistency checking or record of which client is adding information or making requests. Conversely, a client has no way of ensuring that the reasoner has not been given additional information (such as additional axioms) since its last communication.
- There is no explicit classification request. The reasoner will decide when it is appropriate to, for example, build a classification hierarchy of concepts. This may happen after each `TELL` request, alternatively the reasoner may choose to defer the classification until absolutely necessary, or even when there is a lull in traffic.

The specification essentially consists of an XML Schema [18] describing the expressions of the concept language, the available tell and ask operations along with the expected responses and administrative information.

Note that the specification is not intended as a "database system" for knowledge bases. It is simply a protocol that exposes the *reasoning services* provided by a DL reasoner (hence the presence of a number of restrictions as described below, such as the absence of retraction).

[3] `http://www.networkinference.com/`

3 Protocol

Level 0 uses HTTP [11] as the underlying transfer protocol. In this respect, we borrow from other initiatives such as SOAP[4] and XML-RPC[5] which have both built messaging protocols using XML on top of HTTP. The use of HTTP allows client (and server) developers to use existing libraries for implementation[6]. We are not strongly wedded to the use of HTTP as the underlying protocol. A richer mechanism may be adopted in the future. For this Level 0 specification, however, it suffices.

Request Clients communicate with a server through the use of HTTP `POST` requests. The body of the request must be an XML encoded message corresponding to a DIG request as described below. `Content-Type` is `text/xml`, and the `Content-Length` must be specified and must be correct.

The server will use the root element of the message body to determine the message type (i.e. identification, management, ask or tell).

Response Unless there is a low-level error, the server should return `200 OK`. As with requests, the `Content-Type` is `text/xml` and `Content-Length` must be present and correct. The body of the response must be an XML encoded message corresponding to a DIG response as described below.

Persistent Connections The HTTP specification supports the use of persistent connections, which allow requests to be pipelined. This can allow a single TCP connection to be used for multiple requests without waiting for each response. It is envisaged that DIG reasoners implementing this specification will support persistent connections.

Why not SOAP? The interface is using XML over HTTP. Why not just use SOAP? The protocol is intended to be as light weight as possible – by using XML/HTTP, we do not have to worry about any of the SOAP container aspects. In addition, the intention is not to pass *objects* across the interface, but simply messages (e.g. strings). The use of SOAP may ultimately help to integrate into a web service framework, but there is nothing in the current approach that precludes a migration to an alternative protocol such as SOAP.

4 Reasoner Identification

An aspect lacking from the original CORBA-FaCT specification was the ability to identify which reasoner was actually behind the interface. This is particularly important when we may have a number of different reasoners supplying conforming interfaces.

[4] `http://www.w3.org/TR/SOAP/`

[5] `http://www.xmlrpc.com/`

[6] Although this should not be a prime motivation for the use of the protocol, building on top existing work is likely to improve the chances of the DIG Interface being used.

Reasoner Capabilities Ideally, we would expect all reasoners implementing the specification to support the entire concept language and tell/ask functionality. In reality, this is unlikely to be the case in the short term, and some reasoners may choose not to implement, for example, support for concrete domains. In order to cope with this, along with information regarding their identification, a reasoner should also supply details of the language which it supports. This will enable clients to decide whether or not the reasoner will be of use, or guide the clients as to the questions that they can ask of the reasoner.

Such "introspective" descriptions of tools and services are crucial to supporting dynamic component and service discovery as is being pursued in areas such as the Grid.

In the current specification this capabitility information is rather primitive, and essentially amounts to a list of the concept forming operators, tell assertions and queries supported. In the long term, it would be desirable to extend this, for example being able to represent constraints such as whether particular combinations of operation are allowed.

It is assumed that all reasoners will support primitive concepts and roles.

5 Knowledge Base management

A DIG reasoner can deal with multiple knowledge bases. URIs are used in order to identify the different knowledge bases. When a request is made to a reasoner to create a new knowledge base, the reasoner (if successful) will return to the client a URI which the client can then use to identify the knowledge base during TELL and ASK requests (see Sections 6 and 6). Knowledge base URIs are guaranteed to be unique, thus a KB URI is valid for that reasoner only – making a request to another reasoner with the same URI will result in an error. The use of unique URIs also allows us to sidestep some of the issues relating to multiple clients. If a client chooses not to share a KB URI with another client, then the client can be sure that it is the only one interacting with the KB[7]. Different clients of the same reasoner, however, may be able to share knowledge bases by sharing URIs – however it is then the clients' responsibility to manage and coordinate this sharing.

There are two `MANAGEMENT` requests.

- A request for a new knowledge base. The response will return the URI that the reasoner has allocated for the KB (if successful);
- A request to release a knowledge base. In this case, the client should supply the URI of the KB to release. Once a knowledge base has been released, any requests made using the URI should result in an error.

Note that the URIs referred to above simply provide a handle that identifies the particular incarnation of the KB in the particular reasoner.

[7]This is not entirely the case, as a malicious client *could* choose to try sending messages with random URIs to a running reasoner. There is a (very small) chance that such a client could get lucky and hit on a URI which is in use, but this potentional situation is unlikely enough for us not to be concerned with it at this point.

Of course, richer access mechanisms involving, for example, authentication, are likely to be necessary for wider deployment of reasoning services for real-world applications in networked environments. We are aware that our current interface specification (explicitly) ignores this issue in its current form.

6 Message Formats

Communication with the reasoner is via messages which encode the `TELL` and `ASK` functions supported by the reasoner.

Concept Language DIG's concept language is based on $\mathcal{SHOIQD}_n^-$, that is a description logic that includes the standard boolean concept operators ($\sqcap$, $\sqcup$, $\neg$), universal and existential restrictions, cardinality constraints, a role hierarchy, inverse roles, the one-of construct and concrete domains. $\mathcal{SHOIQD}_n^-$ was chosen as it is rich enough to support reasoning over DAML+OIL and OWL-DL. Reasoning for Semantic Web applications is seen as a prime use for DL reasoners in the near future.

Version 1.1 provides rather restricted support for concrete domains, however. Integers and strings are provided, along with concept expressions for minimum, maximum, value equality and ranges. Linear inequations are *not* provided, nor are named concrete objects, although these may be introduced in a later version of the schema. Ranges can be asserted for attributes using assertions like `rangeint` or `rangestring`. If no range is supplied, attributes have integers as their range by default.

Tell Syntax A `TELL` request must contain in its body a `tells` element, which itself consists of a number of tell statements. TELL requests are monotonic – i.e. once information has been told to a knowledge base, it can never be retracted or removed. The only such option available is to release the knowledge base (see Section 5) and then start again. A `TELL` request must be made in the context of a particular knowledge base (which is identified via an attribute of the enclosing `tells` element).

The order of tell statements is unimportant.

The response to a tell will be a `response` message containing either an `ok` element, signifying that the statements were received and interpreted correctly, or an `error` element which may include an optional error code, message and detailed explanation. In addition, an `ok` message may contain warnings about the tells received. For example, the FaCT reasoner will happily process knowledge bases where primitive concepts are used without being explicitly introduced. However, it may be useful to warn the user if this has happened, as this may indicate a spelling or typographical mistake.

Ask Syntax An `ASK` request must contain in its body an `asks` element, which itself consists of a number of ask statements. Each ask statement must have an attribute `id` which supplies a unique identifier for the query (within the context of the particular collection of queries). This allows the presentation of multiple queries in one request, which may in turn allow the reasoner to optimise the processing of these queries. Each `asks` element must also have an attribute that identifies the knowledge base that the queries are being posed against. The value of this attribute should be a URI which identifies a KB within the reasoner.

Response Syntax The schema contains a description of the responses expected of the server to `ASK` requests. The response to an `ASK` request must contain in its body a `responses` element, which itself consists of a number of responses – one for each query in the `ASK`. Each particular response must have an attribute `id` which corresponds to the identifier of a submitted query.

In general, responses to concept queries such as a request for all parents will return sets of sets of concept names, each set being a collection of synonyms.

Interface Granularity One key issue with the specification of an interface is the granularity of the operations supplied. For example, a criticism of the CORBA-FaCT interface was that in order to construct a concept hierarchy (a common task for applications), the client had to make many requests to the server. This was undesirable due to the overhead involved with communication.

This becomes less of an issue when multiple requests are permitted in a single communication. For example, in order to determine the classification hierarchy, a client need only make two requests: one to determine the concepts in the hierarchy (for example through the use of a `<descendants>` or `<allConceptNames>` request, and one to determine all the immediate children of those concepts. This information is then sufficient to recreate the hierarchy without further communication with the reasoner.

7 Implementations

Two implementations of reasoners supporting the DIG protocol are currently available. RACER [8] and FaCT [9] both support DIG 1.1.

The OilEd tool [3] has been altered to use DIG as its primary mechanism for interacting with a reasoner. This allows users to select FaCT or RACER as the reasoner with which to classify and consistency check their DAML+OIL and OWL ontologies.

In addition, a prototype instance store [5] has been implemented that uses DIG to access a DL reasoner. Again, this has allowed us to experiment with alternative reasoners, an invaluable facility when testing.

8 Future Work

The interface described here is, of course, simply a first step towards the provision of reasoners that can be deployed in a component-based architecture. There are many areas in which the specification could be extended. The assumptions described in 2 could be relaxed, in particular to support multiple or stateful clients. The concept language could be extended to cover more expressive DLs. Alternatively we can investigate the provision of a SOAP based interface for better integration in a web service setting.

[8] Available from: `http://www.fh-wedel.de/~mo/racer/`.

[9] Available from: `http://www.cs.man.ac.uk/fact`.

Acknowledgements The authors would like to thank the other members of the Description Logic Implementation Group including Ian Horrocks, Sergio Tessaris, Peter Patel-Schneider, Volker Haarslev and Heiner Stuckenschmidt for their contributions to initial discussions. This work was supported by OntoWeb (EU grant IST-2000-25056) and Wonderweb (EU grant IST-2001-33052).

References

[1] Joint US/EU ad hoc Agent Markup Language Committee. Web ontology language, reference version 1.0. `http://www.daml.org/2001/03/daml+oil-index.html`.

[2] Franz Baader, Diego Calvenese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook.* Cambridge University Press, 2003.

[3] S. Bechhofer, I. Horrocks, C. A. Goble, and R. Stevens. Oiled: a reason-able ontology editor for the semantic web. In *Proceedings of KI2001*, volume 2174 of *LNAI*, pages 396–408, Vienna, September 19-21 2001. Springer-Verlag.

[4] S. Bechhofer, I. Horrocks, P. F. Patel-Schneider, and S. Tessaris. A Proposal for a Description Logic Interface. In *Proceedings of DL'99*, 1999.

[5] S. Bechhofer, I. Horrocks, and D. Turi. A Description Logic Instance Store. Submitted to: 2003 International Workshop on Description Logics - DL'03, 2003.

[6] S. K. Bechhofer, C. A. Goble, A. L. Rector, W. D. Solomon, and W. A. Nowlan. Terminologies and Terminology Servers for Information Environments. In *Proceedings of STEP97*, pages 484 – 497, London, UK, 1997. IEEE Computer Society.

[7] T. Berners-Lee. *Weaving the Web.* Orion Business Books, 1999.

[8] Mike Dean, Dan Connolly, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Web ontology language, reference version 1.0. `http://www.w3.org/TR/owl-ref/`.

[9] Sean Bechhofer et al. An informal description of standard oil and instance oil. `http://www.ontoknowledge.org/oil/downl/oil-whitepaper.pdf`, 2000.

[10] E. Franconi and G. Ng. The i.com Tool for Intelligent Conceptual Modelling. In *Proceedings of KRDB'00*, Berlin, Germany, August 2000.

[11] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and Berners-Lee T. Hypertext Transfer Protocol – HTTP/1.1. `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

[12] V. Haarslev and R. Möller. Description of the RACER System and its Applications. In *Proceedings of DL'01*, Stanford, USA, August 2001.

[13] I. Horrocks. FaCT and iFaCT. In *Proceedings of the International Workshop on Description Logics (DL'99)*, pages 133–135, 1999.

[14] D. L. McGuinness. Ontological issues for knowledge-enhanced search. In *Proc. of FOIS-98*, 1998.

[15] P. F. Patel-Schneider and B. Swartout. Description logic specification from the KRSS effort, 1993.

[16] M. Uschold and M. Grüninger. Ontologies: Principles, methods and applications. *K. Eng. Review*, 11(2):93–136, 1996.

[17] G. van Heijst, A. Schreiber, and B. Wielinga. Using explicit ontologies in KBS development. *Int. J. of Human-Computer Studies*, 46(2/3):183–292, 1997.

[18] World Wide Web Consortium. XML Schema. `http://www.w3.org/XML/Schema`.