Implementation experience with the DIG 1.1 specification

Ian Dickinson
Digital Media Systems Laboratory
Bristol

HPL-2004-85

10th May, 2004*

Email : ian.dickinson@hp.com

description logic reasoners; interface; jena; semantic web; OWL

Description logic reasoners are becoming more widely used for reasoning about resources on the Semantic Web. To allow client tools to interact with different reasoners in a standard way, a common standard interface is highly desirable. The DIG Description Logic Interface is a proposed standard that provides just this capability. In the process of implementing a DIG adapter for the Jena Semantic Web toolkit, we have noted a number of problems and shortcomings in the DIG standard, version 1.1. This report summarises these issues, and makes specific suggestions as an input to the next version of the DIG standard.

* Internal Accession Date Only

Approved for External Publication

Implementation experience with the DIG 1.1 specification

Ian Dickinson
Semantic and Adaptive Systems Dept
HP Labs Bristol
Filton Road, Stoke Gifford, Bristol, UK, BS34 8QZ
mailto:ian.dickinson@hp.com

Abstract

Description logic reasoners are becoming more widely used for reasoning about resources on the Semantic Web. To allow client tools to interact with different reasoners in a standard way, a common standard interface is highly desirable. The DIG Description Logic Interface is a proposed standard that provides just this capability. In the process of implementing a DIG adapter for the Jena Semantic Web toolkit, we have noted a number of problems and shortcomings in the DIG standard, version 1.1. This report summarises these issues, and makes specific suggestions as an input to the next version of the DIG standard.

1 Introduction

Description logics [1] are a class of logical formalism in which certain restrictions are imposed on the expressiveness of the logical language in return for well-established upper-bounds on the complexity of proving theorems using those logics. Description logic theory has thus provided a firm foundation for a number of implementations of description logic reasoners: computer programs that take description logic formulae as input, and deliver theorems about those formulae (such as whether they are logically consistent) as a result. Despite the relative simplicity of the proof method for such logics, practical description logic reasoners are the subject of much research and engineering effort in order to produce reasoners that given acceptable computational performance on realistic problems.

Description logics (typically DL's) are currently of much interest in the Semantic Web community, particularly owing to the recently defined W3C standard Web Ontology Language OWL [4]. Given increasing quantities of ontology data expressed in OWL, semantic web applications will increasingly need access to sophisticated DL reasoners. For developers of semantic web applications, having a convenient means to access different standard DL reasoners would be extremely beneficial. Such a standard also aids the DL implementers, as they can concentrate on better engineering their reasoner, without needing to support the general panoply of representation services needed by application writers. Fortunately a nascent form of such a standard exists: the DIG DL reasoner interface specification [2] from the Description Logic Implementation Group[1]. Although strictly speaking the acronym DIG identifies the group and not the standard, the documentation uses the acronym DIG to refer to the interface specification itself. For brevity and consistency, we follow the same convention in the rest of this paper.

Jena 2 [5] is a popular Java toolkit developed by HP Labs to provide convenient developer support for a wide range of semantic web applications. While Jena includes a rule-based reasoner that is able to provide some of the semantic entailments of the OWL language for a given set of ontology data, it does not attempt to provide the capability of a DL reasoner. We have therefore implemented a plug-in adapter to allow a Jena programmer to utilise a DL reasoner with a DIG interface. This exercise has revealed a number of weaknesses in the DIG standard, which should be addressed in future revisions of the standard. This report has two aims: to report on our implementation experience with DIG, particularly noting the problems that

[1] http://dl.kr.org/dig/

arose, and to make suggestions as input to the next round of revising the DIG specification. It is not an objective to provide a tutorial on the use of the DIG interface itself.

The remainder of this report is structured as follows: section 2 describes our approach in connecting DIG reasoners to Jena. Many of the problems we have discovered have arisen in systematically building translators between DIG's representation language and the RDF representation of OWL. Section 3 details the various issues that have arisen during our development process, and presents our recommendations for future actions to improve the DIG specification. Section 4 concludes.

1.1 A overview and brief history of DIG

The current release of DIG is version 1.1, dated Feb 2003 [2]. DIG was primarily the result of a collaboration between two teams of leading DL engine implementers, those responsible for FaCT [7;8] and RACER [6]. From its genesis, DIG's requirements were primarily driven by the goals of providing access to the capabilities of these two engines, and by the kinds of reasoning tasks that were of interest to the respective research teams. This colours the emphases placed on DIG: for example, FaCT has very weak capabilities for reasoning about instances of classes, while it is very strong in reasoning about class and property hierarchies. Therefore the instance reasoning capabilities of DIG are correspondingly less well-developed. Similarly, both reasoners are typically used in batch mode: a knowledge base is loaded, which may consume considerable computational resources, and then is repeatedly queried. This leads DIG itself to have a batch-oriented flavour, lacking capabilities for incremental changes to the knowledge base (other than strictly monotonic asserts). For the current uses of DIG, this is not a problem. However, as additional DL reasoners are created by other research groups and companies these restrictions are likely to seem more like barriers.

1.1.1 DIG summary

There is not space in this short technical report to provide full details of DIG. Readers who are unfamiliar with DIG 1.1 are referred to the overview [3] or the specification itself [2]. However, to set the context for the remainder of this report, we summarise below the principal features of DIG 1.1.

To provide maximum portability, DIG 1.1 defines a simple XML encoding to be used over an HTTP interface to a DL reasoner. The DL engine thus must provide basic HTTP support (though in fact only the HTTP POST verb is currently used), and be able to parse and generate XML content. While this is quite similar to a web-services approach, the DIG designers explicitly chose not to use web-services machinery (SOAP, WSDL, etc) per se. For our Java implementation this turned out to be convenient, as Jena already includes the Xerces[2] XML library, and basic HTTP support is built-in to the `java.net` library. Thus no additional software libraries were needed to access a remote DIG reasoner.

Each interaction with a DIG reasoner is initiated by a DIG client, which POST's one or more actions encoded using the DIG XML schema. The server will then respond with a combination of an appropriate HTTP response code and, where appropriate, the results of the action or actions similarly encoded in XML.

There are three classes of action that the DIG schema provides for use by clients: management operations, tell operations to make assertions into the reasoner's knowledge base (KB), and ask operations to query the KB.

The DIG management operations are summarised in table 1, while tables 2 to 4 summarise the concept encodings available, and the tell and ask verbs respectively.

[2] See http://xml.apache.org/xerces2-j/index.html

XML element name	Description
`getIdentifier`	Get from the reasoner a description of itself, including name, version number, and a list of the DIG concept language elements supported by the reasoner.
`newKB`	Request the reasoner to allocate a new knowledge base and return the unique identifier for that KB. This KB identifier is used by subsequent operations to indicate which of multiple KB's is being updated or queried.
`releaseKB`	Request the reasoner to release the named KB and free resources associated with it. Subsequent asks or tells to that KB name are in error.

Table 1: DIG management operations

XML element name	Description
`top`	The universal concept (like `owl:Thing`)
`bottom`	The empty concept (like `owl:Nothing`)
`catom`	Introduces a concept (i.e. class) name

`and`
`or`
`not`
	Boolean combinators for concept expressions, corresponding to intersection, union and complement in description logic.

`some R E` `all R E`	Existential and universal property restrictions
`atmost n R E` `atleast n R E`	Maximum and minimum cardinality restrictions (note that the number is encoded as a `num` attribute, not sub-element). Note also that the E parameter, denoting a concept expression, allows qualified cardinality restrictions, a language feature of DAML+OIL that was left out of OWL
`iset`	A class expression that is exactly the given named individuals.
`stringmin, stringmax, stringequals, string-range, intmin, intmax, intequals, intrange`	Limited set of concrete domain value expressions
`ratom`	A named role (object property in OWL)
`feature`	A named functional role (functional object property in OWL)
`inverse`	A role expression denoting the inverse of a given role
`attribute`	A named attribute (datatype property in OWL)
`chain`	A linear composition of features (no equivalent in OWL)
`individual`	A named individual

Table 2: Concept encoding terms

XML element name	Description

`defconcept`
`defrole`
`deffeature`
`defattribute`
`defindividual`
	Concept introduction verbs. Some reasoners may warn if concepts are used before being introduced by a `defXX` verb.

`impliesc C1 C2`	Concept implication, every C1 is also a C2 (equivalent to `rdfs:subClassOf` in OWL)
`equalc C1 C2`	Concept equality (equivalent to `owl:sameClassAs`)
`disjoint C1 C2 .. Cn`	Mutual disjoint-ness between the given concept expressions.
`impliesr R1 R2` `equalr R1 R2`	Role implication and equality; analogous to class implication and equality
`domain R E` `range R E`	Role R has as domain or range the concept expression E
`rangeint A` `rangestring A`	Attribute A has as range the concrete datatype `int` or `string`.
`transitive R`	Role R is transitive
`functional R`	Role R is functional
`instanceof I C`	Individual I is an instance of concept expression C (equivalent to `rdf:type` in OWL)
`related I R I2`	Individual `I` is related to individual `I2` by role R.
`value I A V`	Individual `I` has concrete value V for attribute A.

Table 3: Tell verbs

XML element name	Description

`allConceptNames`
`allRoleNames`
`allIndividuals`
	Query for a list of all named concepts, roles and individuals

`satisfiable C`	Query whether concept C is satisfiable (i.e. can ever have any instances)
`subsumes C1 C2` `disjoint C1 C2`	Query whether C1 subsumes, or is disjoint from, C2

`parents`
`children`
`ancestors`
`descendants`
	Query for the immediate sub- or super- concepts for a given concept (parents/children), or the closure of all super- or sub- concepts (ancestors/descendants)

`equivalents`	Query for all of the concepts equivalent to a given concept

`rparents`
`rchildren`
`rancestors`
`rdescendants`
	Query for the sub- and super-concepts in the role hierarchy, by analogy with the class hierarchy

`instances C`	Query for all instances of concept C
`types I`	Query for all concepts that I belongs to

`instance I C`	Test whether instance `I` is an instance of concept `C`.
`roleFillers I R`	Query for those individuals that are related to `I` through `R`
`relatedIndividuals R`	Query for all individual pairs that are members of `R`
`toldValues I A`	Query for the values of attribute `A` of individual `I`

Table 4: ask verbs

2 Connecting DIG reasoners to Jena

This section outlines the basic design and implementation that allows Jena users to use a DIG reasoner with the Jena RDF API.

The central object that holds RDF data in Jena is the `Model`. `Model` has a rich API, with many convenience functions for RDF programmers. Internally, however, this rich API is problematic for connecting to different kinds of stores (in-memory data structures, persistent databases, LDAP servers, etc). Thus in Jena 2 we introduced a much simpler abstraction `Graph`, which has only a few methods that need to be implemented to wrap a given source of RDF triples. Model is then an adapter for some Graph. A second advantage of introducing the Graph abstraction is that it gives a natural way of providing a certain class of RDF inference services. In many RDF-based languages, such as OWL, the semantics of the language entail the presence of additional triples in the graph of some base data closed under inference. As a simple example, consider graph G which contains exactly two triples:
`:x rdfs:subClassOf :y` and
`:y rdfs:subClassOf :z`.

The inference closure of G, G_i, contains at least the following additional information
```
:x rdf:type rdfs:Class
:y rdf:type rdfs:Class
:z rdf:type rdfs:Class
:x rdfs:subClassOf :z
```

In Jena 2, the inference graph is an implementation of the `Graph` interface, but which wraps some other `Graph` providing the base data, and presents a graph that appears to contain the entailed triples as well as the asserted triples. The Java interface `InfGraph` is a specialisation of `Graph` that wraps a graph of base assertions with a reasoner.

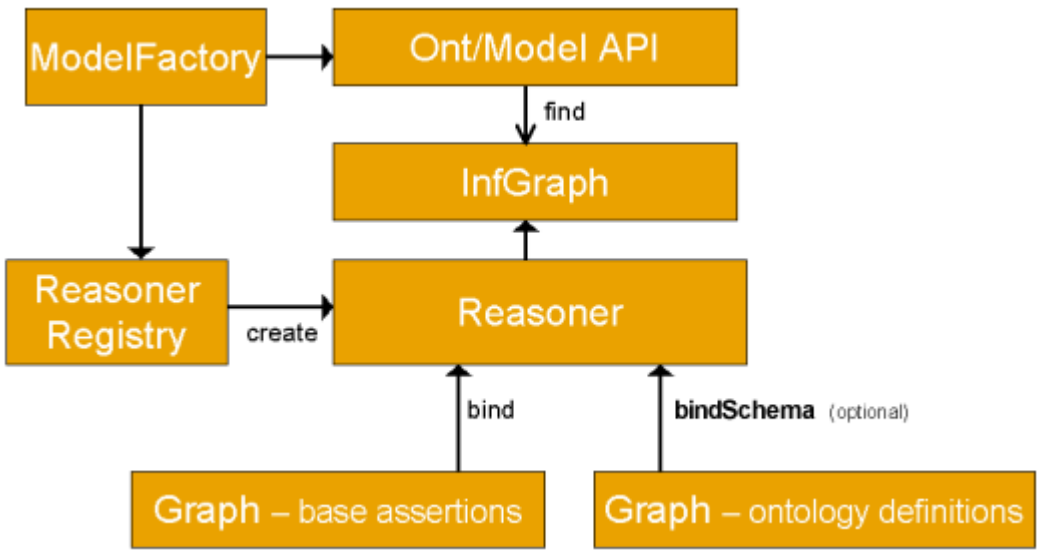

Figure 1: Jena inference schematic

This then provides the natural extension point for integrating a DIG reasoner. A `DIGInfGraph` is the new graph object that is presented by the `Model` adapter. We have to provide a specialisation of `InfGraph` that knows that it is presenting a DIG reasoner, because we must specify the connection details for the remote DIG reasoner. A DIG reasoner is the object that performs the work of translating between RDF triples on one side, and the tells to and asks from the remote DIG reasoner on the other. We discuss `DIGReasoner` in more detail below.

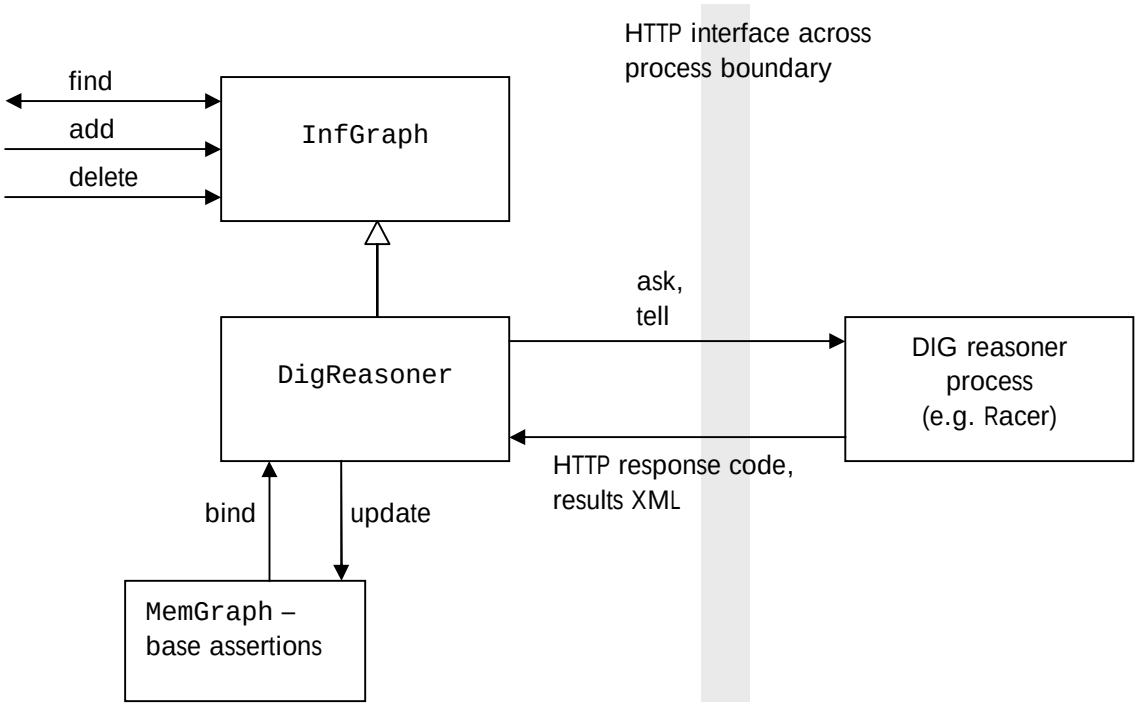

Figure 2: Jena - DIG connection schematic

The other components from Jena's inference machinery are used in the standard way. For example, the reasoner registry is able to create instances of a Jena `InfGraph` bound to a DIG reasoner.

The principal problem we have to deal with in connecting Jena's graph abstraction to a DIG reasoner is not structural, however. As mentioned above, the use of plain XML and HTTP makes the mechanics of the connection straightforward. The problem is to translate between an RDF encoding of an OWL (or DAML, but in this paper we stick to OWL for simplicity) ontology in RDF, and the terms in the DIG concept language. This is easiest to illustrate with an example. In table 5, we show a collection of RDF triples encoding a fragment of an OWL ontology, and the corresponding DIG translation.

RDF/XML	DIG concept language
<owl:Class rdf:ID="A"> <owl:complementOf> <owl:Class rdf:ID="B" /> </owl:complementOf> <rdfs:subClassOf> <owl:Class rdf:ID="C" /> </owl:Class>	<?xml version="1.0"?> <tells xmlns=http://dl.kr.org/dig/lang xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance xsi:schemaLocation="http://dl.kr.org/dig/lang http://potato.cs.man.ac.uk/dig/level0/dig.xsd"> <defconcept name="http://example.org/foo#A" /> <defconcept name="http://example.org/foo#B" /> <defconcept name="http://example.org/foo#C" /> <impliesc> <catom name="http://example.org/foo#A" /> <catom name="http://example.org/foo#C" /> </impliesc> <equalc> <catom name="http://example.org/foo#A" /> <not> <catom name="http://example.org/foo#B"/> </not> </equalc> </tells>

Table 5: sample RDF/XML to DIG translation

The way we accomplish this is essentially to define a simple translator as a lookup table of LHS patterns over sets of RDF triples, and an RHS generator of the appropriate DIG concept language. For example, the pattern "`* rdf:type owl:Class`" matches a generator that introduces `defconcept` terms into the tell verb. At the present time, this lookup table is hand-generated. It would be an interesting experiment to try using a parser generator that would consume a DCG defining the mappings between DIG and RDF, and thus generate the mapping table from a more declarative source. We do, however, define an extensive set of unit tests that compare source expected XML documents to the output of the translation process for a given input set.

A tricky issue with this translation process is when to invoke it. Clearly, if a Jena application loads a set of base data into a local graph, then binds that graph to the DIG reasoner, we do the translation of the entire model to a DIG KB at one time. More difficult is when the updates occur incrementally, because partial information may be transmitted to the reasoner. Consider the following triples being added to a knowledge base (using N3 format for brevity):

```
ns:A rdfs:subClassOf ns:B.
ns:A rdf:type owl:Class.
```

Were these triples to be translated incrementally, some DIG reasoners might issue a warning or error that A is an unknown type, since the `defconcept` would follow the `impliesc` axiom, rather than vice-versa. To avoid this, we would need both to (a) be able to recognise when a set of triple-adds was complete as a unit, and (b) be able to re-order the triples to be translated so that the translations are presented to the DIG reasoner in an appropriate order. The current implementation has solutions to both of these problems, though they have not yet been extensively tested in practise.

The current usage model assumes that Jena is the gatekeeper to all RDF and OWL content. However, a valid alternative approach is that a central DIG reasoner is pre-loaded with the contents of the ontology in question, and then a Jena application connects to that reasoner and presents the DL contents as an RDF model. In principle this is quite possible, but it has not been properly tested and unanticipated problems may arise. This too will be corrected shortly.

3 Problems and recommendations

In the process of arriving at the solution described in section 2, we encountered a number of problems with the DIG interface to RACER (the DL engine we have been testing with). A few of these problems were straight bugs in RACER's DIG implementation, which have now been corrected by the RACER implementers. Other problems were intrinsic to the DIG specification itself. In this section of the report we summarise these problems, as a set of observations and suggestions to any future round of improvements to the DIG specification document.

3.1 General problems

3.1.1 Ambiguity

While very useful, the DIG document [2] is a research report rather than a complete specification. Some terms are not well-defined, some syntax has to be inferred from the examples rather than stated in normative descriptions. Examples: the difference between feature and attribute is not defined (moreover, it turns out that features were made redundant with the introduction of the `functional` axiom); the value axiom when defining the value of an attribute of an individual does not specify the encoding of the value.

Another example of general ambiguity is that the specification does not detail the responses that should be expected from a given query. Usually these are obvious, but in the case of differing interpretations returned by different DIG reasoners, it would be impossible, currently, to tell whether which is correct with respect to the specification. A particular case that needs more attention is the conditions under which the reasoner should report an error: e.g. exactly what constitutes ill-formed input, etc. A test suite should ideally contain examples of ill-formed input that should cause the reasoner to raise particular errors.

A further issue is a general lack of regularity. For example, concept names are introduced with `<catom>`, role names are introduced with `<ratom>`, while individual names are introduced with `<individual>`. Also irregular, from the XML viewpoint, is the treatment of top and bottom. These have their own element forms, which means that code that handles `conceptSet` return values is needlessly complicated by having to handle `<top>` and `<bottom>` together with `catom` values. Making top and bottom special names for `catom` elements would improve the conceptual simplicity of both the underlying model and the code that has to process it.

Recommendation: a pass of refactoring and editing of the document, with a view to being an unambiguous standard for other implementers. Ideally the specification should be augmented with a normative set of conformance tests.

3.1.2 Result and results

There are two wrappers for returned results `<response>` indicates a single value, `<responses>` indicates more than one (see figures 10 and 14 of [2]). However, the size of the results set (one or greater) is easily determined from the XML tree, so the distinction between result and results is not useful.

3.1.3 Querying attribute values

Consider the following DIG tell message (edited for clarity, e.g. xmlns declarations removed):

```
<tells>
    <defconcept name="http://example.org/foo#A"/>
    <defattribute name="http://example.org/foo#p"/>
    <defindividual name="http://example.org/foo#a0"/>
    <instanceof>
        <individual name="http://example.org/foo#a0"/>
        <catom name="http://example.org/foo#A"/>
    </instanceof>
    <value>
        <individual name="http://example.org/foo#a0"/>
```

```
            <attribute name="http://example.org/foo#p"/>
            <ival>3</ival>
        </value>
        <value>
            <individual name="http://example.org/foo#a0"/>
            <attribute name="http://example.org/foo#p"/>
            <ival>1</ival>
        </value>
        <value>
            <individual name="http://example.org/foo#a0"/>
            <attribute name="http://example.org/foo#p"/>
            <ival>2</ival>
        </value>
    </tells>
```

The next segment shows query against this KB, and the current response[3]. In Dig 1.1, this is the only way to query against attribute values:

```
    <asks>
        <toldValues id="q0">
            <individual name="http://example.org/foo#a0"/>
            <ratom name="http://example.org/foo#p"/>
        </toldValues>
    </asks>

    <responses>
        <ival>2 </ival>
        <ival>1 </ival>
        <ival>3 </ival>
    </responses>
```

This response raises several problems. Firstly, there's no aggregate element, analogous to `conceptSet` or `individualSet`, in other DIG response types. Secondly, the basic idea of `toldValues` seems to be mixing two different concepts: one is the dual of `roleFillers` for attributes, and the other is querying the asserted values and not the deductive closure.

It is incongruous to make consulting the asserted facts, excluding the deductions, only available to attribute values and not to other query types. It would also seem prudent to allow the DIG syntax to support reasoners that can perform reasoning on concrete domain values, as future DIG-capable reasoners may indeed do so.

Recommendations: querying for attribute values needs a proper query verb, and an aggregate element to group the results (such as `valueSet`). Querying the told values vs. the deductive closure should be factored out as general option to queries, perhaps as an optional attribute on the `asks` element.

3.1.4 Use of HTTP error codes

While, in general, we support the choice of the DIG designers to make minimal assumptions about the choice of communications protocol, for example by not specifying a SOAP interface, we also recognise that the flexibility to adapt to other communications platforms is important. For designers who wish to adopt the DIG specification as an XML content language, but use, for example, SOAP or an agent communication language (ACL) as the communications substrate, DIG is overly restrictive in specifying that error conditions are signalled using HTTP error codes. It would be better for the DIG specification to be self-contained at the XML level.

---

[3] Using RACER, but only as an exemplar. The problem is not that RACER is buggy, but that DIG does not have the syntax to express the results appropriately.

Recommendation: the DIG specification should be extended so that the error conditions that are currently specified by HTTP return codes (see appendix A of the DIG specification) are instead returned as XML elements in the return message. Use of HTTP error signalling should be reserved for use by clients using HTTP to access the server, purely to signal errors intrinsic to HTTP communications. In particular, HTTP error codes should not be used to indicate parse or recognition errors with the XML content of the DIG message.

3.2 Issues arising from translating OWL RDF to DIG

3.2.1 Unique name assumption

When RACER and FACT were first created, it was common for DL engines to make the unique name assumption (UNA). This states that two entities that have different names necessarily refer to different things. Thus, if the DL reasoner was able to infer that two names referred to the same entity, a contradiction was reported and the KB would be deemed unsatisfiable. RDFS, DAML+OIL and OWL, because they are intended for the open world of the Web, explicitly do not make the UNA. Indeed, DAML+OIL and OWL have mechanisms for stating explicitly that two individuals are the same (q.v. `owl:sameAs`) and different (q.v. `owl:differentFrom`). This equivalence on individuals cannot be translated into DIG at the moment (while there is `equalc` and `equalr` for class and role equivalence, there is no `equali` for individuals), and in any case no DIG capable reasoners at the time of writing (April '04) would be able to handle `equali` without contradiction because of the UNA. A forthcoming release of RACER is intended to make the UNA switchable, precisely to better support OWL. It is not clear whether a future version of FaCT will do so.

Recommendations: DIG needs to be extended to include equality of instances (i.e. something like `equali`, and synonyms to be added to the `individualSet` response syntax). The reported reasoner capabilities (in the identifier response) should be extended to report whether the reasoner supports the unique name assumption. Optionally, there should be a way of configuring the UNA through DIG, for those reasoners that support it[4].

3.2.2 Anonymous resources

A core feature of RDF and its derivative languages is the anonymous resource, or bNode. DIG does not allow any resources to be anonymous. The way that the Jena DIG adapter handles this is to build a table of Skolem names, constants that are guaranteed not to occur elsewhere in the model, and use that table to map between the bNode identifier and the Skolem name that is passed to DIG. We use URN's constructed from UUID's for this purpose. However, such URN UUID's are also used legitimately in other applications, so there is a danger that URN naming schemes may accidentally clash.

Recommendations: either extend DIG syntax properly to account for anonymous resources, or establish a common URN namespace for Skolem names that stand for RDF bNodes. The latter is certainly the easier option: it would model anonymous RDF resources as named objects in a DIG reasoner, but using a name that was unlikely to clash with existing names. This could be recognised by RDF clients as a proxy for an anonymous resource; such clients could then perform an appropriate transformation to true RDF bNodes.

3.2.3 `hasValue` restriction

OWL and DAML allow the definition of a class expression as a restriction on a property p where the value of p is a certain value, either an individual or a concrete data value. For example, the class `IansBrothers` is the set of individuals for whom the value of the `brother-of` property is `:Ian`. DIG does not have a direct equivalent for `hasValue`. Sean Bechhofer suggested the following alternative formulation for a restriction on property p that it `owl:hasValue I`:

---

[4] Adding a general parameter-setting and getting might be a simple way of providing this capability in a generic way.

```
<some>
   <role name="p"/>
   <iset>
      <individual name="I"/>
   </iset>
</some>
```

For datatype properties (i.e. attributes in DIG), we have something like:

```
<intequals val="2">
   <attribute name="p" />
</intequals>
```

This works for the few datatypes (int and string) that are built-in to DIG, but will not work for general typed literals.

Recommendations: (1) Direct support for `hasValue` restrictions in the DIG syntax, including `hasValue` operating over concrete domains (as permitted by OWL). (2) Extend DIG syntax to allow XSD datatypes to be specified for concrete domain values, so that reasoners that are capable of working with datatypes have sufficient information to do so.[5]

3.2.4 Inverse functional property

OWL and DAML have the notion of a property that has a unique domain value for a given range value. This is the opposite of a functional property, so OWL's terminology is inverse functional property (DAML uses the term unambiguous property). There is no direct translation for inverse functional property in DIG. One way to encode "`:p rdf:type owl:inverseFunctionalProperty`" is to create axioms stating that property p' is the inverse of p, and that p' is functional. The translator will have to create a Skolem name for the inverse property p' if none is already defined in the model, and this may produce confusing output for users, especially if an inverse of p is later declared in the KB.

Recommendation: DIG should be extended to include direct support for inverse functional property axioms.

3.2.5 Different from and all-different

OWL and DAML both allow individuals to be declared to be explicitly different. This is implied by the UNA, so there is no explicit syntax for it in DIG. Sean Bechhofer suggested the following encoding, that works for both `owl:differentFrom` and `owl:allDifferent`:

```
<disjoint>
  <iset>
   <individual name="i1"/>
  </iset>
  <iset>
   <individual name="i2"/>
  </iset>
</disjoint>
```

Recommendation: Although this encoding works, Sean Bechhofer also points out that it may inhibit reasoners that have machinery to efficiently handle `allDifferent` and `differentFrom` axioms from recognising situations where that support can be employed. Therefore, we recommend an explicit encoding for both axiom types.

---

[5] Suggestion (2) is from Sean Bechhofer.

3.3 Missing components

Some desirable verbs or queries are simply missing from the DIG 1.1 specification.

3.3.1 Testing the global consistency of the KB

There is no query verb that tests that the overall KB is consistent[6].

3.3.2 Equivalent roles

There is a query verb `<equivalents>` for querying the concepts that are equivalent to a given concept, but there is no corresponding verb for querying the roles that are equivalent to a given role.

3.3.3 Querying property axioms

A role may be declared functional, transitive, etc, but there is no means for a DIG client to query the KB to discover the axioms for a given property. This will be particularly pertinent for Jena when a Jena model can be connected to an existing reasoner that already has the knowledge base loaded. Moreover, sometimes the characteristics of a role, such as being symmetric, can be inferred from the KB. This would be good deduction for the reasoner to be able to report to the client.

Sean Bechhofer recently commented[7] that there are likely to be a number of queries of a KB that are not supported currently in DIG, but which are likely to be useful in the general case when viewing a DIG reasoner as a general knowledge server. This has not been a focus of DIG in the past, but may become increasingly important in the future.

3.3.4 Type testing for concept, role and individual atoms

Given a name, a client may wish to know whether that name is currently defined as a concept, role, attribute or individual. For concepts, roles and individuals the only way to achieve this is to list all of the appropriate names (e.g. asking `<allConceptNames />`), and then testing whether the given name is one of those. This is potentially very inefficient for large KB's. There appears to be no means at all to test whether a given name corresponds to an attribute.

One way to rectify this is to provide a set of unary ask predicates to test whether a given name is a concept, role, etc. An alternative is a predicate that tests the type of a name, and returns one of `role`, `attribute`, `concept`, `individual` or `undefined`.

3.3.5 Testing and retrieving disjoints and equivalents in class hierarchy

The ask language contains the following verbs, both of which are useful:

```
<disjoint>C0 C1</disjoint>    returns true/false
<equivalents>C0</equivalents> returns a conceptSet
```

That is, for disjoint classes you must ask whether C0 is disjoint from C1, but for equivalent classes you must ask which concepts are equivalent to C0. The duals of these asks should be added to the ask language:

```
<disjoints>C0</disjoints>        returns a conceptSet
<equivalent>C0 C1</equivalent>   returns true/false
```

---

[6] Note that Racer's DIG interface provides the non-standard verb `<consistentKB />` to test the global consistency of the KB.

[7] Personal communication.

A particular problem that is caused by not having the second form of `equivalent` is that it is unnatural to query as to whether a given named class is equivalent to a concept expression[8], and moreover it is not possible to ask if two concept expressions are equivalent (for example is "A ∩ B `owl:equivalentClass` A ∩ C"?)

3.3.6 Listing role fillers for all roles on an individual

The query `<roleFillers> I R </roleFillers>` allows a DIG client to retrieve all values for role R on an individual `I`. It is also important for many applications to list all the role-fillers for all of `I`'s roles, i.e. to provide a query similar to `roleFillers` but in which R is not known a priori.

4 Conclusion

This report has summarised some of our experiences building an adapter to connect Jena inference models to DIG reasoners. DIG is a good start towards a standard interface to a description logic reasoners. Having such a standard will be of great value to the semantic we community, allowing semantic web tools to interoperate much more effectively.

The current version of the DIG specification suffers somewhat from not having been widely implemented yet. Our attempt to build, ab initio, a DIG reasoner client exposed a number of shortcomings in version 1.1 of the DIG specification. We hope that these observations will contribute to improving future versions of the DIG standard, and help to make DIG a successful for range of DL reasoner interfaces.

5 Acknowledgements

The author gratefully acknowledges the assistance of Sean Bechhofer, Ralf Möller and Volker Haarslev in responding to questions and bug reports during the development of the Jena DIG adapter. Thanks also to Dave Reynolds of HPLabs for comments on an early draft of this paper, and to Sean Bechhofer for detailed comments on version 0.3.

6 References

1. Baader F, Calvanese D, Mcguinness D, Nardi Daniele, Patel-Schneider P. ((eds)). The Description Logic Handbook: Theory, Implementation and Applications Cambridge University Press , 2003.

2. Bechhofer, Sean. The DIG Description Logic Interface: DIG/1.1. 2003.
Available from: `http://dl-web.man.ac.uk/dig/2003/02/interface.pdf`

3. Bechhofer, S., Möller, R., & Crowther, P. "The DIG Description Logic Interface". In: Proc. 2003 International Workshop on Description Logics (DL2003). CEUR Workshop Proceedings, 2003.
Available from:
`http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-81/`

4. Bechhofer, Sean, van Harmelen, Frank, Hendler, Jim, Horrocks, Ian, McGuinness, Deborah L., Patel-Schneider, Peter F., and Stein, Lynn Andrea. OWL Web Ontology Language Reference. W3C. 2004.
Available from: `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`

5. Carroll, Jeremy J., Dickinson, Ian, Dollin, Chris, Reynolds, Dave, Seaborne, Andy, and Wilkinson, Kevin. Jena: Implementing the Semantic Web Recommendations. (HPL-2003-146) HPLabs Technical

[8] The client must query for the equivalents of the class expression, and look for the named concept in the return value.

Report. 2003.
Available from: http://www.hpl.hp.com/techreports/2003/HPL-2003-146.html

6. Haarslev, V. & Möller, R. "Description of the RACER System and Its Applications". In: DL-2001 2001 International Description Logics Workshop. CEUR Workshop proceedings, 2001.
Available from:
http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-49/HaarslevMoeller-132start.ps

7. Horrocks, I. The FaCT system. 2003.
Web site: http://www.cs.man.ac.uk/~horrocks/FaCT/

8. Patel-Schneider, P. F. & Horrocks, I. "DLP and FaCT". In: Murray, N. V. (ed) Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'99. Springer Verlag (Lecture Notes in Artificial Intelligence), 1999. pp. 19 – 23.