Parsing OWL DL: Trees or Triples?

Sean Bechhofer
Department of Computer Science
University of Manchester
Manchester, M13 9PL
UK
seanb@cs.man.ac.uk

Jeremy J. Carroll
Hewlett-Packard Labs
Bristol, BS34 12QZ
UK
jjc@hpl.hp.com

ABSTRACT

The Web Ontology Language (OWL) defines three classes of documents: Lite, DL and Full. All RDF/XML documents are OWL Full documents, some OWL Full documents are also OWL DL documents, and some OWL DL documents are also OWL Lite documents. This paper discusses *parsing* and *species recognition* – that is the process of determining whether a given document falls into the OWL Lite, DL or Full class. We describe two alternative approaches to this task, one based on abstract syntax trees, the other on RDF triples, and compare their key characteristics.

Categories and Subject Descriptors

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Representation languages*; D.3.4 [**Programming Languages**]: Processors—*Parsing*

General Terms

Algorithms, Languages, Performance

Keywords

Semantic Web, OWL, Parsing, RDF

1. INTRODUCTION

In 1999, van Harmelen and Fensel [16], argued:

> No matter how nice any Knowledge Representation language is as proposed by the AI community, [. . .] the order of precedence is the other way round: *how well* can AI concepts be fitted into the markup languages that are widely supported on the Web, [. . . *our emphasis*]

This question is answered with OWL, the Web Ontology Language. The underlying AI concept is Description Logic [1], the Web markup language is RDF [12].

This paper discusses *how well* the triple oriented RDF abstract syntax can encode the more conventional tree structured syntax for description logics.

The OWL Semantics and Abstract Syntax Recommendation [14] normatively defines OWL. Two different semantics are given, one for the RDF triples, the other for the OWL abstract syntax trees (corresponding to a mainstream description logic syntax such as in [1]). Mapping rules are specified linking the trees with the triples.

For this to work, for OWL to make sense, it has to be possible to switch between the triples and the trees. A minimal task is that of being a syntax checker [7] – which involves classifying the tree, given the triples.

During the development of OWL, some doubt was expressed as to whether this was possible. An implementor reading the documentation gets a shock when they realize they have to run the nondeterministic mapping backwards.

We describe two different implementations, one based around the trees, the other on the triples.

1.1 What is OWL Syntax?

The Web Ontology Language (OWL) [9] defines three classes of documents: Lite, DL and Full. All RDF/XML documents are OWL Full documents. Some OWL Full documents are also OWL DL documents, and some OWL DL documents are also OWL Lite documents. The characterisation of OWL DL and OWL Lite is essentially *syntactic* in nature. That is, the relevant rules define structural manipulation, rather than the semantic rules that give interpretation of structures.

The first structural rules are those defined by RDF/XML syntax [3], which gives a set of rules for converting an RDF/XML document into an RDF graph [12].

This paper is concerned with the further rules, found in the OWL Semantics and Abstract Syntax [14] (S&AS), which then characterise the RDF graphs that are in OWL DL and OWL Lite.

Syntax checking can be seen to have a number of uses. Implementors may choose to target a particular OWL sublanguage. For example, OWL DL has been chosen to yield a language for which inference is *decidable*. An application targeting OWL DL will need to know whether ontologies are amenable to inference using the choice of DL semantics and reasoning techniques. At a more prosaic level, anecdotal experience suggests that many ontologies are OWL Full not through explicit choice, but rather through errors (see Section 2.4) – for example missing type triples may point to typographical errors in the ontology source. A syntax checker can prove useful in finding such errors.

1.2 Terminology

A *parser* takes an input document and returns an abstract syntax tree (which can then be classified).

A *recognizer* (or species recognizer) takes an input document and indicates if it belongs to OWL DL or OWL Lite.

A parser can easily be transformed into a recognizer, but not vice versa.

An *OWL Syntax Checker*, as defined in the OWL Test Cases Recommendation [7], is a recognizer for OWL DL and OWL Lite.

1.3 Two Approaches

This paper presents two different systems reflecting two different approaches to OWL syntax. The more conventional, WonderWeb parser, constructs an abstract syntax tree and checks its well-formedness. This approach is also used by the other OWL Syntax checkers that reported during the OWL Candidate Recommendation, such as OWLP and Pellet.[1] The other, the Jena checker, is a recognizer, which is strongly triple oriented, depending on a pre-transformation of the grammar and mapping rules to be a triple-centric grammar with no reference to abstract syntax trees.

2. OWL SYNTAX

A document is in OWL DL, if it is an RDF/XML document for which the corresponding graph conforms to the rules for OWL DL.

The rules for OWL DL are defined constructively in S&AS. An abstract syntax is defined, that describes a set of parse trees. Each of these parse trees can then be converted into one or more RDF graphs using nondeterministic mapping rules. This is shown in table 1.

An OWL syntax checker, has to, at least implicitly, do this process backwards – i.e. take an RDF graph, invert the mapping rules, and hence find a corresponding abstract syntax tree. If there is one, then the document is in OWL DL, otherwise it is in OWL Full.

If more than one graph corresponds to a parse tree, then these graphs have the same semantic interpretation. Moreover, more than one parse tree may correspond to the same RDF graph, in which case the two trees have the same semantic interpretation.

2.1 The Abstract Syntax

The abstract syntax rules are described in section 2 of S&AS [14].

These are fairly conventional looking BNF [10] rules:

⟨*ontology*⟩ ::= ‘`Ontology(`’ [⟨*ontologyID*⟩] { ⟨*directive*⟩ } ’)’

⟨*axiom*⟩ ::= ‘`Class(`’ ⟨*classID*⟩ ⟨*modality*⟩ . . . ’)’
| ’DatatypeProperty(’ ⟨*datavaluedPropertyID*⟩ . . . ’)’

⟨*individual*⟩ ::= ’Individual(’ [⟨*individualID*⟩] . . .
{ ’type(’ ⟨*type*⟩ ’)’ } { ⟨*value*⟩ } ’)’

The principle novelty is that these rules describe abstract syntax trees, and not a document. There is no intent that the terminal leaves of the tree be read off to form a text string. Thus the abstract syntax is a set of trees, defined by a BNF.

The trees defined by these rules are not quite the parse trees according to the BNF, but structural trees defined by the ‘(’ and ‘)’ in the terminals in the rules. A simple rule like:

⟨*fact*⟩ ::= ⟨*individual*⟩

is not made explicit in any corresponding abstract syntax tree.

2.2 The Mapping Rules

The mapping rules are described in section 4 of S&AS [14]. A typical mapping rule looks like:

Individual(

$$\begin{array}{ll} \text{value}(pID_1 v_1) & \xrightarrow{\_:x} \_:x\ T(pID_1)\ T(v_1). \\ \ldots\text{value}(pID_k v_k)\ ) & \ldots\_:x\ T(pID_k)\ T(v_k). \end{array}$$

This shows that an abstract syntax tree matching the left hand side, can be transformed into triples as given on the right hand side. The functor $T(\cdot)$ is used to show recursive application of the mapping rules. A node is returned from such a recursive application that is used within the triples on the right hand side of the rule.

We show the node to be returned (a ‘main node’ in the terminology of S&AS), as a superscript above the arrow of the rule.

The mappings of the substructures are shown on the right hand side in the same order as the abstract syntax tree on the left. This is important when there are many optional or repeated elements.

2.3 OWL Lite or OWL DL?

Some OWL DL documents are also in OWL Lite. They are those for which there is an abstract syntax tree which uses only the grammar rules from the OWL Lite section of S&AS.

As discussed in the OWL Overview [13], OWL Lite and OWL DL can be partially differentiated by the vocabulary used, for example `owl:unionOf` does not occur in OWL Lite. However, this approximation is wholly inadequate for writing a species recognizer.

There are a number of situations when different constructions in the abstract syntax could yield the same triples. For example, axioms **[A]** and **[B]** in Table 2 both yield the same RDF triple shown in the table, where `T(restriction( p cardinality(0) )` is the bnode created to represent the restriction. Note however, that in this case, **[A]** may be part of an OWL Lite ontology while **[B]** may not as it involves a disallowed expression in an axiom. This particular example illustrates that species recognition between DL and Lite is not simply a case of checking vocabulary – we must also examine how the vocabulary has been *used*. An ontology is in OWL Lite if there is *some* abstract tree fitting the OWL Lite conditions that yields the given triples under the mapping rules.

2.4 Error Classification

There are, in general, two ways in which an RDF graph may fail to correspond to an OWL Lite or DL ontology.

- There is an OWL Lite or DL ontology in abstract syntax form which maps to a superset of the given triples but some of the triples have been forgotten and are not in the graph.
- The ontologies in abstract syntax form that map to the triples or any superset of the triples violate some of the restrictions for membership of the OWL Lite or DL subspecies. (This includes the case where there are no such ontologies).

We might (loosely) describe the first as *external* errors, and the second as *internal* errors.

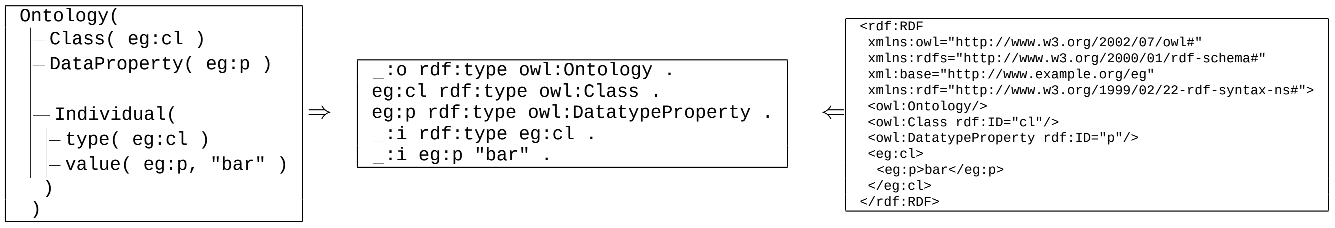
Examples of *external* errors include:

- Using a URI reference in a particular context (e.g. as the subject of an `rdfs:subClassOf` triple) without including an appropriate explicit type triple;
- Malformed syntactic constructs, e.g. a node typed as an `owl:Restriction` that is not the subject of an `owl:onProperty` triple;
- Using the wrong vocabulary, e.g. `rdf:Property` instead of the more specific `owl:ObjectProperty` or `owl:DatatypeProperty`;

Examples of *internal* errors include:

- Violation of the rules concerning separation of classes, individuals and properties (in DL and Lite we require that these interpretations are disjoint);
- The use of expressiveness outside the scope of the species – for example using an `owl:unionOf` in an OWL Lite document;

[1]See `http://www.w3.org/2001/sw/WebOnt/impls` for details.



```
Ontology(
  Class( eg:cl )
  DataProperty( eg:p )

  Individual(
    type( eg:cl )
    value( eg:p, "bar" )
  )
)
```

⇒

```
_:o rdf:type owl:Ontology .
eg:cl rdf:type owl:Class .
eg:p rdf:type owl:DatatypeProperty .
_:i rdf:type eg:cl .
_:i eg:p "bar" .
```

⇐

```
<rdf:RDF
 xmlns:owl="http://www.w3.org/2002/07/owl#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 xml:base="http://www.example.org/eg"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 <owl:Ontology/>
 <owl:Class rdf:ID="cl"/>
 <owl:DatatypeProperty rdf:ID="p"/>
 <eg:cl>
  <eg:p>bar</eg:p>
 </eg:cl>
</rdf:RDF>
```

OWL Abstract Syntax [14]   RDF Graph [12]   RDF/XML Syntax [3]

Mapping Rules

Table 1: Overview of Definition of OWL Syntax

	Abstract Syntax	RDF triples
[A]	`Class ( a complete restriction( p cardinality(0) ) )`	`a owl:equivalentClass T(restriction( p cardinality(0) ) )`
[B]	`EquivalentClasses ( a restriction( p cardinality(0) ) )`	

Table 2: Axioms and OWL Lite

- Redefining the reserved vocabulary (e.g. those things in the OWL, RDF and RDF(S) namespaces).
- A directed cycle of blank nodes is usually an internal error (see section 2.6.2)

Errors concerning structure sharing, the use of blank nodes, may fall into either category depending on the exact error, (see section 2.6.2).

It may often be the case that "missing" triples are simply due to an omission, rather than a desire to use expressiveness outside the scope of OWL DL. For example a URI may only be used in a "class" context, but without an explicit triple. In this case, the document will be OWL Full, but a "fix" may be applied to the document (effectively adding the missing triple). We must be careful when applying such fixes that the original intention of the document is not altered, but such a facility is likely to prove useful. We return to this issue in Section 9.

2.5 Imports

A number of difficult issues in parsing and recognition relate to imports. Species validation must be done on the *imports closure* of the ontology – this effectively involves retrieving URIs that are the object of an `owl:imports` triple and adding any triples from an RDF graph found there to the current RDF graph.

Validation cannot be performed *locally*, i.e. without first calculating the imports closure[2], as it may be the case that required type triples are actually present in the imports. For example in test *imports-005*[3], the required triple typing the imported URI as an ontology (required in OWL DL) is actually contained in the imported ontology. Similar situations can arise with classes and properties.

It can also be that case that an ontology imports an RDF graph from a URI where the imported graph is in OWL Full, but the importing ontology still remains in OWL Lite. This would be the case where, for example, the imported ontology asserts `a rdf:type b` without explicitly typing `b` as a class. We return to this issue in Section 8.

2.6 Blank Nodes

A number of conditions regarding valid OWL syntax relate to blank nodes (or bnodes) in the RDF graph. A blank node is a node that is not a URI reference or a literal – it is a unique node that can be used in one or more RDF statements, and has no globally distinguishing identity. A key point is that bnodes cannot be referred to from outside the document that we are processing.

Bnodes generated by the mapping correspond to:

- Anonymous classes, e.g. arbitrary class expressions such as intersections, unions and enumerations.
- Restrictions, e.g. existential quantifications over properties.
- Anonymous individuals, e.g. *John's brother*.

The mapping rules [14] state:

Bnode identifiers here must be taken as local to each transformation, i.e., different identifiers should be used for each invocation of a transformation rule.

Thus whenever an expression such as `intersectionOf (Person Male)` is used in an ontology, the mapping creates a *new* bnode corresponding to that expression. In general, no sharing of bnodes is permitted – each bnode can participate as the object of at most one triple.

There are, however, two cases where a blank node corresponding to an expression can be used in more than one place – when the translation results from an `EquivalentClasses` or `DisjointClasses` axiom.

2.6.1 Bnodes in axioms

When anonymous class expressions are used in `EquivalentClasses` or `DisjointClasses` axioms, the mapping rules permit the re-use of the resulting bnodes produced. However, this reuse is only allowed within the context of the triples produced by the mapping from that particular axiom.

For an equivalent classes axiom:

`EquivalentClasses(` d_1 `...` d_n `)`

the mapping requires the production of a set of `owl:equivalentClass` triples that form an (undirected) connected graph over the nodes produced by translating each d_i.

For a disjoint classes axiom:

`DisjointClasses(` d_1 `...` d_n `)`

the mapping requires the production of a set of `owl:disjointWith` triples s.t. each node produced by translating a d_i is related to every other d_j for $i \neq j$ as either subject or object in a `owl:disjointWith` triple (forming an `owl:disjointWith` *clique*).

[2]See Section 5.3. of S&AS.

[3]`http://www.w3.org/2002/03owlt/imports/Manifest005.rdf`

We can see that in the above cases, bnodes corresponding to a translation from an anonymous class expression may participate in a number of triples. They may not, however, participate in any triples that do not correspond to those generated by the mapping rule applied to the axiom.

In addition, the `SubClassOf` axiom, may introduce a blank node that is the subject of an `rdfs:subClassOf` triple, and such a blank node cannot be the object of any triple.

2.6.2 Bnode Summary

In summary, blank nodes must fit at most one of the following cases:

1. Be the subject or object of any number of `owl:equivalentClass` triples
2. Be the subject or object of any number of `owl:disjointWith` triples (in which case a further check must be applied)
3. Be the subject of an `rdfs:subClassOf` triple.
4. Be the object of an `rdfs:subClassOf` triple.
5. Be the object of some other triple.

Hence, a graph may have an internal error concerning a blank node which is in more than one of these categories, or is involved in two triples in cases 3, 4 or 5. Or it may have an external error, concerning the blank nodes involved with `owl:disjointWith` triples which may not form a clique.

In addition blank nodes may not form directed cycles, except in cases 1 and 2.

3. SYSTEM DESCRIPTIONS

In the following sections we provide overviews of the two systems discussed in this paper – these will be referred to as WonderWeb and Jena.

3.1 WonderWeb

WonderWeb[4] is an EU IST FET project concerned with "Ontology Infrastructure for the Semantic Web". As part of the work of WonderWeb, an API for OWL Ontologies has been developed, providing a collection of (Java) interfaces allowing the representation and manipulation of OWL ontologies. A detailed description of the rationale behind the API is given in [2], but put briefly, the API insulates application developers from the vagaries of concrete syntax, and provides a higher level view of the objects (classes, properties, axioms etc) in an OWL Ontology. The structure of the data model in the API is largely based on the OWL Abstract Syntax [14].

The WonderWeb API also aims to separate the functionality that one might require when working with OWL ontologies. Aspects of functionality such as:

- change (addition/removal)
- serialization
- parsing/deserialization
- inference

are all considered separately, allowing implementations to be clear about the services they provide. The codebase of the API (including a species validator as described here) is available for download as an open source project[5].

Of course, insulating applications from concrete syntax is all well and good, but it is clear that mechanisms for parsing and serializing from/to concrete representations are vital for real-world applications. To this end, a parser for OWL ontologies represented in RDF/XML has been produced. The parser takes an RDF/XML document and attempts to produce a corresponding abstract syntax tree.

The WonderWeb OWL API is particularly targeted at the OWL DL and Lite species (a research agenda of the project is the use of Description Logic reasoners with OWL). The ability to recognize when a particular document is in a species (and is thus amenable to the appropriate reasoning techniques) is a key requirement, and the WonderWeb parser performs species recognition as part of its parsing process.

3.2 Jena

Jena[6] [6] is an open source semantic web developers kit, principally developed at HP Labs.

It provides APIs for manipulating RDF graphs. The Ontology API provided for OWL and DAML ontologies, while abstracting from the underlying RDF graph, does not attempt to totally hide or replace it.

Moreover, the OWL support is intended as OWL Full support with reasoning support for cases not included in the OWL DL syntactic subset. The Ontology API within Jena has explicit OWL Full support handling the polymorphism that can occur between classes and individuals, datatype properties and object properties, etc.

Thus the syntax checker requirements are an add on to the ontology support rather than a prerequisite. The typical user requirement which may be addressed is to verify that a graph is within OWL DL before saving it to be exported. There is no requirement to produce abstract syntax trees.

In keeping with the RDF-centric philosophy of Jena, the syntax checker operates in a triple oriented fashion, and hence depends upon a different triple oriented expression of the grammar of OWL DL. This is produced in a precompilation stage, akin to the well known compiler-compiler technique [11]. In this stage, which occurs while the system is being built, the abstract syntax rules and the mapping rules are analyzed in detail. A set of syntactic categories for nodes in a graph and a table of triples linking these syntactic categories is produced. This table of triples is used as the grammar at runtime.

The Jena syntax checker's principle task is to find a mapping from the nodes in the graph to the syntactic categories such that every triple appears in the transformed grammar.

4. THE WONDERWEB PARSER

The WonderWeb parser takes an RDF graph and attempts to build an abstract syntax tree. The basic strategy employed is as follows.

1. Identify named objects: classes, properties
2. Identify axioms asserted in the ontology. In the course of identifying axioms, we may need to translate nodes corresponding to class expressions.
3. Translate anything that's left. Again this may require the conversion of class expressions occurring as the subject of `rdf:type` triples.

During this process, we keep a note of those triples that have been "used", e.g. those that are identified as being the result of the application of the mapping rules to a particular construct. By doing this we can identify any triples that are "left" after we have constructed all classes, properties and the axioms concerning those

[4]`http://wonderweb.semanticweb.org`

[5]`http://sourceforge.net/projects/owlapi`

[6]`http://jena.sourceforge.org`

classes. These unused triples are then interpreted as facts about individuals in the ontology.

Identifying named objects is simply a case of finding all those (named) nodes that are the subject of an `rdf:type` triple where the object is `owl:Class`, `owl:ObjectProperty` or `owl:DatatypeProperty`[7]. In these cases, optional triples may also be present – for example if we have `a rdf:type owl:Class`, the rules also allow the addition of `a rdf:type rdfs:Class` (even though the additional triple adds no extra semantic information).

Once those objects are identified, we can translate axioms concerning them. For example, for any triples of the form: `p owl:inverse q` we check that `p` and `q` are instances of `owl:ObjectProperty` and add `q` to the list of inverses held by `p`. If `p` and `q` are not instances of `owl:ObjectProperty`, and we are simply interested in recognizing OWL DL and OWL Lite ontologies, we can stop at this point as we now know that the RDF graph cannot correspond to a OWL DL or Lite ontology.

Other axiom types such as `rdfs:subProperty` are dealt with in a similar manner. A more interesting translation task is when the axiom deals with a bnode representing a class expression as in Figure 1. In this situation, we first identify the node which is the object

```
<owl:ObjectProperty rdf:about="#p">
  <rdf:domain>
    <owl:Class>
      <owl:complementOf rdf:resource="#A"/>
    </owl:Class>
  </rdf:domain>
</owl:ObjectProperty>
```

Figure 1: Complex Domain Expression

of the triple. This is then translated to a class expression by a case analysis on the triples in which the node appears as subject. In general, there should be a single such triple, with its predicate determining the form of the expression produced. The presence of more than one such triple indicates an OWL Full ontology. Recursive translations may be necessary if the expression includes nested expressions.

Lists are used to represent the operands in expressions such as intersections. Such lists are translated by collecting all the nodes that appear in the list (as the object of an `rdf:first` triple) and forming a new expression using the translations of those nodes. The well-formedness of the list (each node in the list should be the subject of exactly one `rdf:first` and `rdf:rest` triple) is also checked during this process.

4.1 Axioms and OWL Lite

As discussed in Section 2.3, care needs to be taken when handling axioms concerning classes. The strategy employed here is to translate any `rdfs:subClassOf` and `owl:equivalentClass` triples to "class definitions" whenever possible. For example, with a triple `a rdfs:subClassOf x`, where `a` is a named node (as opposed to a bnode), then we attempt to construct an object corresponding to:

```
Class( a partial Tx )
```

where `Tx` is the translation of `x` to a class expression. Similarly, if we encounter `a owl:equivalentClass x` then this is translated to

```
Class( a complete Tx )
```

(but see later discussion on handling blank nodes). This strategy ensures that OWL Lite ontologies are produced whenever possible.

[7]we must also identify instances of `owl:OntologyProperty` and `owl:AnnotationProperty`, but space precludes us from providing a detailed exposition of the parsing process here.

4.2 Handling Imports

The use of `owl:imports` allows us to refer to RDF graphs held at separate locations. Although the validation process is performed over the imports closure of the graph, it can be useful to try and process each chunk separately. It is often the case when we have `onto1 owl:imports onto2` that the statements at `onto1` are intended to form a single `Ontology`, while those at `onto2` form another[8]. A formal definition of this difficult due to the inexpressiveness of RDF – we cannot represent the fact that particular assertions belong in a particular `Ontology` (see Section 8). The parser attempts, wherever possible, to perform this "chunking" (based on the physical locations of the triples) and builds individual `Ontology` objects corresponding to each RDF graph retrieved from a particular URI. This is done by recursively calling a new parse on an imported URI.

In order to do this successfully though, we need to pass information between the parsing processes, in particular recording whether URIs have been correctly typed. This then allows us to deal with situations where a URI is used in a class context in `onto1` and has the appropriate type triple in `onto2` (or vice versa).

In adopting this approach to imports, we need to relax our handling of typing somewhat. In the `owl:inverse` example above, the *local* information may not be enough to determine whether the properties are of the required types as the triple regarding the type of `p` may be in an imported ontology. In this case, we make an assumption that the types are appropriate, and check at the end of the complete parsing process that any such assumptions made have been discharged (e.g. we really encountered the appropriate typing triple). Assumptions are also passed to any recursive parse along with type information. If assumptions remain at the end of the parse, required type triples were missing, signifying an OWL Full ontology.

Note that the grouping of statements into separate ontologies has no effect on the semantics of importing ontology, in terms of the *entailments* that hold.

4.3 Blank Nodes and Structure Sharing

In addition to flagging "used" triples, the parsing process also flags bnodes as they are translated to expressions or used in lists. If a flagged bnode is encountered in a translation, this indicates that structure sharing has occurred, the document is OWL Full, and appropriate action can be taken.

Cases involving equivalence and disjointness axioms (see Section 2.6.1 require special handling. In order to check whether equivalence axioms are well-formed, we do the following.

1. Gather together all nodes that participate in triples involving `owl:equivalentClass`;
2. Partition these nodes into sets, where two nodes `a` and `b` are in the same set iff there exists a triple `a owl:equivalentClass b` or `b owl:equivalentClass a`.

Each of these equivalence classes can now be translated. If the size of the set is 2 and the triple that induced the set is of the form `a owl:equivalentClass x` where `a` is a named node, then we translate to a class definition (see Section 4.1). Otherwise we translate to an `EquivalentClasses` axiom.

The conditions regarding disjointness are more complicated. The rules for `DisjointClasses` axioms tell us that an axiom: `DisjointClasses(` $d_1\ d_2\ \ldots d_k$ `)` is translated to a collection of nodes, one for each expression in the equivalence, and a number of `owl:disjointWith` triples, such that every node in the collection is connected to every other node by at least one triple (in either direction). This may lead to blank nodes being used in more than one

[8]E.g. the `food` and `wine` examples from the OWL Guide [15].

place and participating in many triples. We apply the following strategy to `owl:disjointWith` triples.

1. Gather together all nodes that participate in triples involving `owl:disjointWith`;
2. While there are bnodes in the collection of nodes that we have not already dealt with, do the following:
   (a) Pick a bnode n that we haven't already dealt with.
   (b) Gather together all the nodes $n_1, n_2, \ldots n_k$ that can be reached from n via a path that consists of `owl:disjointWith` triples, and which does not pass through a named class node – in other words the traversal stops when we reach a named node. Include n in this collection.
   (c) In order for the graph to be in OWL DL, the subgraph formed from these nodes considering `owl:disjointWith` edges must be fully connected: every node must have an edge to every other node. If this is not the case, the graph is not in DL. If it is the case, then we add a `DisjointClasses` axiom using translations of the nodes in the collection formed above.

5. THE JENA RECOGNIZER

The Jena recognizer uses a very different technique. We introduce it with an example, followed by describing the key concept of node categorization, before launching into a detailed discussion.

5.1 An Example of the Approach

Suppose we are given the following three triples in order:

```
_:r owl:onProperty eg:p .
_:r owl:hasValue "a value" .
eg:p rdf:type owl:ObjectProperty .
```

When processing the first triple, we can conclude that it must have come from one of the mapping rules for restrictions, for example:

$$\begin{array}{lcl} \texttt{restriction(}\ \textit{ID} & & \texttt{\_:x rdf:type owl:Restriction .}\\ \quad\texttt{allValuesFrom(} & \xrightarrow{\_:x} & \texttt{\_:x owl:onProperty } T(\textit{ID})\ .\\ \qquad \textit{range}\ \texttt{))} & & \texttt{\_:x owl:allValuesFrom } T(\textit{range})\ . \end{array}$$

Thus `eg:p` must be either a datavaluedPropertyID[9] or an individualvaluedPropertyID, and `_:r` is the node corresponding to some restriction.

When we process the second triple, we already know that `_:r` is a restriction of some sort, and the additional triple tells us that it is a `value(·)` restriction. Moreover, the literal object, tells us that this is a value restriction using the following mapping rule:

$$\begin{array}{lcl} \texttt{restriction(}\ \textit{ID} & & \texttt{\_:x rdf:type owl:Restriction .}\\ \quad\texttt{value(} & \xrightarrow{\_:x} & \texttt{\_:x owl:onProperty } T(\textit{ID})\ .\\ \qquad \textit{value}\ \texttt{))} & & \texttt{\_:x owl:hasValue } T(\textit{value})\ . \end{array}$$

We note that for $T(\textit{value})$ to be a literal, then *value* must be *dataLiteral* and the following abstract syntax rule must have been used:

⟨*dataRestrictionComponent*⟩ ::= 'value(' ⟨*dataLiteral*⟩ ')'

This rule can only fire if ID is a *datavaluedPropertyID*.

Thus, the second triple tells us that `_:r` corresponds to a value restriction on a *datavaluedPropertyID*. If we now return to the first triple, given the new information about `_:r` we now know that `eg:p` is a *datavaluedPropertyID*.

Since the mapping rule only applies to abstract syntax constructs that come from OWL DL we know that the triples are not from an OWL Lite document.

There is nothing more that can be said about either the predicate or the object of either the first or second triples. Thus neither triple will make any further difference to the rest of the processing, and both could be discarded in an incremental recognizer. All that needs to be remembered is the classification of `_:r` and `eg:p`.

When we come to the third triple, we find a *datavaluedPropertyID* as the subject of an `rdf:type` triple, with an `owl:ObjectProperty` object. The mapping rules do not produce such triples, and so this is an internal error (cf. section 2.4).

If we processed the triples in the reverse order, we would have concluded that `eg:p` was an *individualvaluedPropertyID*, (from the third triple), and found the error while processing the first triple, because the grammar does not generate `owl:onProperty` triples linking value restrictions on datavalued properties with *individualvaluedPropertyID*s.

5.2 Node Categorization

The example depended upon an analysis of

- Whether `eg:p` was an *individualvaluedPropertyID* or a *datavaluedPropertyID*.
- What sort of restriction corresponded to `_:r`

We view this as a function from the nodes in the graph to a set of syntactic categories generated from the grammar.

Each uriref node may be a builtin uriref, with its own syntactic category (such as `owl:onProperty`), or a user defined ID, such as a *classID*.

Each blank node is introduced by one of the mapping rules. We hence have one or more[10] syntactic categories for each mapping rule that creates a blank node.

5.3 The Category Refinement Algorithm

The main goal of the algorithm is to determine which category each of the nodes is in.

To make the runtime algorithm simpler, the grammar (including the mapping rules) is preprocessed into a grammar table of triples of syntactic categories.

Two of the entries in this table, relevant to the example are:

```
individualValuedProperty rdf:type owl:ObjectProperty .
literalValueRestriction owl:hasValue literal . (DL)
```

Some of the entries are annotated with actions, for example the second triple sets the not-Lite flag.

Each step in the algorithm processes one triple.

The currently known possible categories for each of the three nodes are retrieved. Each combination of these is tested to see if it is in the grammar table. Such tests allows the elimination of some of the previous possible categories for each node.

If all the possible categories are eliminated, then the graph did not conform to the syntax.

The algorithm is specified in terms of the definition of a function C that assigns a set of categories to each node in the graph.

1. For each blank node n in the graph, set $C(n)$ to the set of all blank categories.
2. For each builtin uriref n in the graph, set $C(n)$ to be $\{n\}$.
3. For other urirefs n in the graph, set $C(n)$ to be the set of all ID categories (classID etc).

[9]The reader will need to refer to an open copy of S&AS [14] during this section!

[10]Sometimes, the combination of the abstract syntax and the mapping rules, is such that the same mapping rule is used for two different abstract syntax constructs. The rule for the value restriction is one, which can be used for both literal values and object values. In such cases, we clone the mapping rule and have one for each abstract syntax construct, giving rise to two syntactic categories for blank nodes.

4. For each node equivalent to `"0"^^xsd:int "1"^^xsd:int` set $C(n)$ to $\{$*liteInteger*$\}$ (for use in cardinality restrictions).
5. For each other node equivalent to `x^^xsd:nonNegativeInteger` set $C(n)$ to $\{$*dlInteger*$\}$.
6. For any typed literal node with user defined type set $C(n)$ to $\{$*uTypedLiteral*$\}$.
7. For each other[11] literal node set $C(n)$ to $\{$*literal*$\}$.
8. For each triple $t =< s, p, o >$ in the graph, *refine*(t), where *refine* is defined as:
   (a) Set $S = C(s)$, $P = C(p)$, $O = C(o)$, to be the set of categories currently associated with the subject, predicate and object of t respectively.
   (b) Set $S' = \{s* \in S | \exists p* \in P, o* \in O$ with $< s*, p*, o* > \in \textit{Grammar}\}$
   (c) If S' is empty then fail.
   (d) Set P' and O' similarly
   (e) If $S \neq S'$ update $C(s) := S'$ and for each t' involving s which has already been processed, *refine*(t')
   (f) Similarly for P' and O'
   (g) If every match from S', P' O' in the grammar table is annotated with the same action, perform that action.
9. Check for missing triples, (i.e. for external errors, see section 2.4).

Since the values of C are strictly monotonic decreasing through the recursive steps 8e and 8f, the algorithm terminates. The actions in step 8g and the final checks in step 9 are discussed in more detail below.

5.4 The Compiler Compiler

The compiler compiler transforms the OWL DL grammar from the form in S&AS [14] to a triple oriented form suitable for the Jena checker.

The input consists of:

- A list of the names of the syntactic categories for urirefs (e.g. classID).
- The abstract syntax (somewhat reformulated)
- The mapping rules (somewhat reformulated).

The output is as follows:

- A list of syntactic categories for nodes (148 categories: 45 for the keywords in OWL, such as `rdf:type`, 14 corresponding to the different uses of user defined urirefs, 83 for different usages of blank nodes, 6 artificial pseudocategories)
- Various groupings of these categories (e.g. all those categories that are involved with `owl:disjointWith`).
- A table of legal triples of syntactic categories, annotated with actions and a DL flag (2486 entries).
- A lookup functions that assign an initial set of syntactic categories to a node

The compiler compiler is written in Prolog, and the grammar and mapping rules have been written in a Prolog form. A detailed discussion can be found in [5].

The rules concerning blank nodes corresponding to descriptions and restrictions are somewhat complicated. There are two natural categories: descriptions being blank nodes with explicit type `owl:Class`, and restriction being blank nodes with explicit type `owl:Restriction`. It is convenient to subdivide these categories into one category per mapping rule.

[11]This rather *ad hoc* list of literal classifications reflects precisely the relevant division in OWL DL syntax. In particular, non-integer XSD literals are treated the same as plain literals.

We saw in section 2.6.2 that blank node usage fell into four cases: in the compiler compiler, this is expressed by converting each of the syntactic categories coming from a mapping rule for descriptions or restrictions into four subcategories, one for each of these cases. Since there are 19 such mapping rules in the grammar used, this accounts for 76 of the 83 blank node categories.

5.5 The Actions

The actions used by the grammar are: the DL actions, for triples which do not occur in Lite; an Object action when the object of this triple is a blank node which must not be the object of any other triple; and the actions FirstOfOne, FirstOfTwo and SecondOfTwo when the subject of this triple corresponds to a construct with one or two components each reflected by a triple in the graph. This triple is the stated component (e.g. `owl:onProperty` is the first of the two components of a restriction). For each of these, the runtime processing remembers the triple as fulfilling the specified role and it is an error if some other triple plays the same role.

The actions are only acted on when all remaining categorizations of the triple require it. In particular, this ensures that the DL action is not invoked until it is not possible to match the triple with only the OWL Lite grammar.

5.6 Pseudotriples

Given the framework of category refinement, some of the other syntactic aspects of OWL can be expressed within it. This is achieved by introducing additional syntactic categories, which are included in the initial assignment of categories to nodes. The table of triples is extended with a further virtual table of pseudotriples using these virtual categories. This table of pseudotriples is a short piece of code rather than actual entries in a table.

By an appropriate choice of which pseudotriples are in the virtual table, global properties can be propagated through the node categories.

The goal with the pseudocategories is that nodes with syntactic defects are marked as being in a pseudocategory. When a triple is processed which addresses those defects then the node is no longer marked as in the pseudocategory.

The final stage of the algorithm searches for all marked nodes and takes appropriate action (such as rejecting the input).

5.6.1 Typing

As an example, most nodes in OWL Lite and OWL DL have to have an explicit type triple in OWL Lite and OWL DL.

In the Jena checker, all relevant nodes have initially category assignment including the category `notype`. This pseudocategory appears in pseudotriples in all three places with arbitrary real categories in the other two places. The key exception is when the predicate is `rdf:type` which typically provides the required type. Such triples do not appear in the pseudotable.

Thus, if there is an appropriate type triple, the pseudocategory is removed from the category assignment for the node by the operation of the refinement algorithm.

5.7 The Final Checks

These checks check external errors, i.e. where needed triples were missing, and some internal errors which were not fully covered elsewhere.

To continue the typing example, the final check is a simple inspection for nodes in the pseudocategory `notype`.

Every blank node in category such as restriction or description which require one or two structural triples, is inspected to verify that such triples have been found.

5.8 The Difficult Cases

Most of the difficult cases are handled using a combination of pseudocategories, pseudotriples and final checks. This is particularly suited to the external errors, which cannot be detected by the refinement algorithm. Some of these cases concern exceptions to the required type triple rule, such as an `owl:Class` being a permitted type for a restriction. Others concern *orphans*, blank nodes that are not the object of any triple: for example, list nodes may not be orphans.

Most directed cycles of blank nodes are prohibited. While these form internal errors, the refinement algorithm cannot detect them. However, using three pseudocategories, it can detect many cases of provably non-cyclic nodes (e.g. a blank node that is the object of a triple whose subject is a URIref or a non-cyclic blank node). The final check then only examines those nodes not already proven to be non-cyclic.

The hardest part is checking the constraint on `owl:disjointWith`. During the refinement algorithm each pair of nodes linked by `owl:disjointWith` is saved in a classical undirected graph G. The final check then verifies the following transitivity constraint on G, which is sufficient for there to be appropriate `owl:disjointWith` cliques:

$$\forall a, c \in V(G)\ \forall \text{ blank } b \in V(G),$$
$$\{a, b\} \in E(G) \wedge \{b, c\} \in E(G) \Rightarrow \{a, c\} \in E(G)$$

6. PERFORMANCE

Performance figures are shown in Table 3. The two systems were run on the OWL Test Cases (resulting in approx 480 single document recognition tasks). In addition, the systems were given a large[12] OWL Lite ontology (the NCI Cancer Ontology[13]) to recognize. Test documents were cached locally to reduce delays due to network access.

These are not intended to be detailed test results, but show the rough performance of the systems. Elapsed time (in seconds) and total memory required are given. Figures are broken down by test size (in terms of triples in the models). Tests were run on a PC running Windows 2000 with a Pentium III 866 MHz processor and 512 Mb memory.

As we can see from the figures, the systems are roughly comparable in time and space required. Jena's better performance on the working group tests can be explained in part by the fact that it does less work and does not produce abstract syntax trees.

7. COMPARISON

Here we compare the two approaches and consider their advantages and disadvantages.

7.1 Jena

The Jena implementation has two main attractions: much of the code is generated from the grammar, and, in principle, the algorithm need only remember relatively small amounts concerning triples that have been processed.

A key defect is that the approach does not generate an abstract syntax tree.

In general, generating code from a grammar using a compiler compiler should make it easy to change. Many changes can simply be copied into the source grammar, and the system is then recompiled. However, the grammar for OWL, particularly the mapping rules, is augmented with English text, which provides additional difficulties that sabotage the simple recompile. Moreover, the treatment in the Jena checker depends on a number of global features of the abstract syntax grammar, such as the assignment of a type to every node. These features are partly there as a result of lobbying by the second author between the first draft of S&AS and the final recommendation. Hence, the approach is fragile to change.

A further advantage of using a compiler compiler is that the core engine is very small, which has facilitated optimization, permitting some incremental processing, see [5].

For the external errors, the Jena code could give elegant error messages. For the internal errors, the code currently computes a minimal subgraph exhibiting the error. The error message then prints this subgraph. This is not user friendly, and is a consequence of the design with a core table driven engine. A way to improve the error messages would be to write additional code that examined the minimal subgraph produced looking for common problems. This would tend to duplicate some of the WonderWeb code.

What would make the Jena code particularly attractive is if the overall design of OWL DL, with an abstract syntax and mapping rules, were duplicated for some other RDF extension. A possible candidate may be any specific ontology, for which the data files would be OWL files using mainly the `fact` directives, and most of the `axiom` directives would be disabled. Moreover, additional restrictions may be applied to the `fact` directives. This would allow an ontology creator to specify a syntactic conformance to that ontology. If the structure of the OWL DL definition were used, then the Jena checker could be recompiled using the new definition. To make this feasible, a significant clean up to the approach taken in S&AS would be needed. In particular, it is not plausible to support *ad hoc* English annotations to the formal rules.

7.2 WonderWeb

The WonderWeb approach results in the construction of an object representing the abstract syntax tree of the Ontology. This has a number of advantages, in particular it can facilitate further manipulations or translations of the ontology, e.g. to an alternative format amenable for processing. This has allowed us to experiment with alternative reasoning strategies for OWL using first order reasoners[14] or logic programming.

In addition, the approach allows us to provide "user friendly" error messages, informing the user *why* their ontology fails to belong to a particular language species. Similarly, fixing particular kinds of errors (such as missing type triples) would be relatively easy.

Actually building the abstract syntax objects does not come without a cost. In "difficult" RDF graphs, say where we have two `owl:Ontology` objects in the graph, we have no way of deciding where to put anything – if we are solely interested in validation (e.g. whether an appropriate ontology can exist), then this is not an issue, otherwise we can only apply heuristics to determine where information should be contained.

As with Jena, the WonderWeb codebase is largely unoptimised and the memory footprint is large (see Section 6). The strategy employed requires the entire RDF graph (or at least an interface allowing query of the entire graph) to be available to the checker. This could, of course, be done using some persistent storage, reducing the memory requirements. Similarly the construction of the abstract syntax objects requires storage, which is currently held in main memory.

The main drawback of the approach, however, is that it is effectively a bespoke or hard-coded solution – the rules for validation

[12]over 500,000 triples.

[13]`http://www.mindswap.org/2003/CancerOntology/`

[14]`http://wonderweb.man.ac.uk/owl/first-order.shtml`

Test size		40	80	120	200	600	1000	1500	3000	NCI
Time	WonderWeb	194	268	284	291	355	394	454	1450	49,200
(ms)	Jena	4	11	15	21	95	155	239	710	301,436
Memory	WonderWeb	102	143	211	220	1,098	1,207	1,848	3,901	281,059
(Kb)	Jena	50	103	151	176	582	714	962	2,060	356,428

Table 3: Performance Figures

are encapsulated in the implementation, both of the parser and the post-parsing validation. Small scale changes to the mapping rules could be accommodated, large scale changes would require a more extensive rewrite. Changes to S&AS as discussed in Section 7.1 are unlikely to be of much benefit here.

Another disadvantage is that the original structure of the RDF is lost – thus the WonderWeb parser and API is not well suited to handling general RDF[15].

8. DISCUSSION

8.1 Imports

It is clear that the handling of `owl:imports` is a crucial aspect to parsing and recognition. AS&S says:

an `owl:imports` annotation also imports the contents of another OWL ontology into the current ontology and requires that an interpretation of an ontology O satisfies the ontology iff it also satisfies all ontologies mentioned in an `owl:imports` directive.

In terms of RDF graphs, the interpretation of imports is that

*[an ontology] is **imports closed** iff for every triple in any element of K of the form `x owl:imports u`, then K contains a graph that is the result of the RDF processing of the RDF/XML document, if any, accessible at `u` into an RDF graph. The imports closure of a collection of RDF graphs is the smallest import-closed collection of RDF graphs containing the graphs.*

There is a tension here between these interpretations of `owl:-imports` – the interpretation in terms of RDF graphs does not necessarily coincide or respect the boundaries of the interpretation in terms of "abstract syntax" ontologies.

This tension is reflected in the two implementations discussed here. The triple based approach used in Jena handles imports in a natural fashion. The abstract syntax approach taken in WonderWeb requires careful handling of imports – in some situations heuristics have to be applied in order to determine exactly where assertions belong, and information must be passed around during parsing of imported ontologies.

8.2 Context

A related issue here is that of containment or *context*. The mapping rules translate an Ontology to a collection of triples. This collection includes triples that relate to the ontology object itself. For example the ontology:

```
Ontology( U
  Class( a ) )
```

yields the following triples:

```
U rdf:type owl:Ontology          (a)
a rdf:type owl:Class             (b)
```

The problem here is that there is no connection between triple `(a)` and `(b)`, other than the fact that they occur in the same graph. The fact that the typing occurs within the context of the Ontology is **not** represented explicitly.

If an `owl:imports` triple occurs, the imported RDF graph is simply added to the existing graph – again though the fact that there is no explicit representation of the *origin* of those statements means that the context has to be handled using heuristics.

The situation is compounded further by the fact that the mapping rules allow for an RDF graph to be the translation of a collection of OWL ontologies. In this case, we cannot even apply heuristics[16] to determine which assertions belong with which ontology. Of course, this difficulty is a direct consequence of the desire to actually reverse the mapping rather than demonstrate that a reverse mapping is *possible* – which is all that is required for recognition. The desire to construct a representation closer to the abstract syntax is, however, not unreasonable.

The link between the physical location of the RDF graph and the URI assigned to the ontology in the abstract is also unclear. For example the ontology:

```
Ontology( http://xyz/things ... )
```

could be mapped to an RDF graph G1 which is then made available at a URI `http://abc/stuff`. A second ontology can now make use of this:

```
Ontology( Annotation(owl:imports http://abc/stuff) ) ... )
```

If this ontology is mapped to an RDF graph G2, calculating the imports closure of G2 actually results in the addition of the G1 into the RDF graph, including the triples referring to `http://xyz/things` as an ontology. This is at best confusing, and is likely to result in OWL Full ontologies due to the lack of appropriate `owl:Ontology` triples.

Carroll and Stickler [8] suggest that the naming of RDF graphs should be promoted to a first class operation, and not handled implicitly through document names. They suggest that this improves the semantics of both the `owl:Ontology` class and the `owl:imports` predicate.

9. FIXING UP OWL

Both systems could be modified to fix errors to transform OWL Full documents into OWL DL documents. Some changes would be semantically sound, others unsound. The sound changes are those where the original document both entails and is entailed by the transformed document, according to the OWL Full semantics. Unsound changes either lose information or add information to the original document (or both).

Example sound fixups are:

- Adding missing type triples for classes and individual valued properties.
- Adding missing type triples with type `owl:Thing` for individuals.

[15]This is no surprise though as it is not intended to be a general RDF API.

[16]For example, lumping all the triples obtained from a single URL together as one "ontology".

- Doing a deep copy of a unnamed class description when it is the object of two triples (soundness depends on a conjecture from Carroll [4]).
- Converting a named restriction into a named class, and an unnamed restriction, linked by an `owl:equivalentClass` triple.

Example unsound fixups that add information, i.e. the transformed document entails the original:

- Adding missing type triples for data valued properties.
- Using a skolemization to resolve bnode problems.

Example unsound fixups that lose information, i.e. the original document entails the transformed document:

- Forgetting either a transitivity constraint or a cardinality constraint on a property with both.
- Doing a deep copy of an unnamed individual which is the object of two triples.
- Take a large DL subset of the RDFS closure of the input graph. This is particularly useful when the OWL vocabulary has been extended. This is easier in Jena than in WonderWeb, since Jena includes an RDFS reasoner.

A further unsound, but potentially useful, fixup is to clone a property that is used both as a data valued property and a individual valued property. One version is used for data values, the other for individual values.

For the external errors both systems know what went wrong and the appropriate place for the fixup is clear. For internal errors, the Jena recognizer faces the same problem as with error messages, in that it finds bad subgraphs without a clear idea of why they are bad. Hence the fixup code would need to be able to analyse such subgraphs to identify the problem. Some errors defy fixup – for example cycles of bnodes in descriptions.

10. CONCLUSIONS

We answer the syntactic aspects of van Harmelen and Fensel's question [16] of "*how well* can AI concepts be fitted into the markup languages on the Web?" with a weak affirmative: well enough, (but it was not easy).

We have demonstrated that it **is** possible to build OWL parsers and recognisers. Moreover, one of our implementations reflects the needs and interests of the AI community, the other those of the Web community. This is a non-trivial exercise, but the information in the OWL document set is sufficient to allow implementors to build recognisers that behave appropriately on the OWL Test Cases [7]. The identification and discussion of issues and hard cases may also prove useful for those wishing to implement OWL-based systems.

Demonstration of implementation experience is a key prerequisite to the endorsement of a recommendation by the W3C. The existence of both the WonderWeb and Jena checkers[17] and the fact that the two implementations decribed here adopt very different strategies to the task can be taken as further evidence that the specification *is* implementable.

To answer the question raised in the title of the paper, both triple- and tree-based approaches are possible. Each has its pros and cons – which to choose depends primarily on the perspective of the application.

[17]Along with other systems such as OWLP and Pellet – see `http://www.w3.org/2001/sw/WebOnt/impls` for details

11. ACKNOWLEDGMENTS

Jeremy Carroll is a visiting researcher at ISTI, CNR in Pisa, and thanks his host Oreste Signore. Sean Bechhofer is supported by the WonderWeb project (EU grant IST-2001-33052) and thanks Boris Motik and Raphael Volz for implementation assistance. The authors would also like to thank Peter Patel-Schneider for useful discussions on tackling the harder cases.

12. REFERENCES

[1] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[2] S. Bechhofer, P. Lord, and R. Volz. Cooking the Semantic Web with the OWL API. In *2nd International Semantic Web Conference, ISWC*, volume 2870 of *Lecture Notes in Computer Science*, Sanibel Island, Florida, October 2003. Springer.

[3] D. Beckett. RDF/XML Syntax Specification (Revised). `http://www.w3.org/TR/rdf-syntax-grammar/`, 2003.

[4] J. J. Carroll. B1 B2 proof [sic]. `http://lists.w3.org/Archives/Public/www-webont-wg/2003Jun/0294/`, 2003. This proof is known to be flawed, the conjecture is open.

[5] J. J. Carroll. Streaming OWL DL. In *European Semantic Web Symposium*, May 2004.

[6] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *WWW2004*, 2004.

[7] J. J. Carroll and J. D. Roo. Web Ontology Language (OWL) Test Cases. `http://www.w3.org/TR/owl-test/`, 2003.

[8] J. J. Carroll and P. Stickler. RDF Triples in XML. Technical Report HPL-2003-268, HP Labs, 2003.

[9] M. Dean and G. Schreiber. OWL Web Ontology Language Reference. `http://www.w3.org/TR/owl-ref/`, 2003.

[10] ISO/IEC. Information technology – Syntactic metalanguage – Extended BNF. Technical Report 14977:1996(E), ISO/IEC, 1996.

[11] S. C. Johnson and R. Sethi. Yacc: a parser generator. In *UNIX Vol. II: research system*. W.B.Saunders, tenth edition, 1990.

[12] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. `http://www.w3.org/TR/rdf-concepts/`, 2003.

[13] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. `http://www.w3.org/TR/owl-features/`, 2003.

[14] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. `http://www.w3.org/TR/owl-semantics/`, 2003.

[15] M. K. Smith, C. Welty, and D. L. McGuinness. OWL Web Ontology Language Guide. `http://www.w3.org/TR/owl-guide/`, 2003.

[16] F. van Harmelen and D. Fensel. Practical Knowledge Representation for the Web. In D. Fensel, editor, *Proceedings of the IJCAI'99 Workshop on Intelligent Information Integration*, 1999.