Cooking the Semantic Web with the OWL API

Sean Bechhofer[1], Raphael Volz[2], and Phillip Lord[1]

[1] University of Manchester, UK
seanb@cs.man.ac.uk,p.lord@russet.org.uk
http://www.cs.man.ac.uk
[2] Institute AIFB, University of Karlsruhe, Germany
rvo@aifb.uni-karlsruhe.de
http://www.aifb.uni-karlsruhe.de

Abstract. This paper discusses issues that surround the provision of application support using OWL ontologies. It presents the OWL API, a high-level programmatic interface for accessing and manipulating OWL ontologies. We discuss the underlying design issues and illustrate possible solutions to technical issues occurring in systems that intend to support the OWL standard. Although the context of our solutions is that of a particular implementation, the issues discussed are largely independent of this and should be of interest to a wider community.

1 Introduction

To realize the vision of the Semantic Web, the Web Ontology Working Group [21] has been chartered to develop a standard language for expressing semantics on the web. The Web Ontology Language (OWL) comprises a standardized syntax for exchanging ontologies and specifies the semantics of the language, i.e. how the syntactic structures are to be interpreted.

However, it is unclear precisely how to slice the pie between the disciplines of syntax and semantics in applications. Support for OWL in applications involves understanding how syntax and semantics interact (i.e., their interface). A number of issues relating to this split continually re-occur in the design of Semantic applications, e.g. in the development of OntoEdit [19], OilEd [3] and KAON [6].

This paper discusses a number of the technical issues encountered when "implementing OWL" and introduces the OWL API, with which we can provide a high-level programmatic interface for both accessing and manipulating OWL ontologies. Besides presenting the underlying design issues we illustrate solutions to these issues in systems that intend to support the OWL standard.

The provision of APIs allows developers to work at a higher level of abstraction, and isolate themselves from some of the problematic issues related to serialization and parsing of data structures. Our experience has shown that application developers can interpret language specifications such as DAML+OIL in subtly different ways, and confusion reigns as to the particular namespaces and schema versions used[1]. The direct use of higher level constructs can also help

[1] **Quiz Question:** Without checking the schemas, can you be sure whether `type`, `comment` and `Property` belong to the RDF or RDF(S) vocabularies?

D. Fensel et al. (Eds.): ISWC 2003, LNCS 2870, pp. 659–675, 2003.

to alleviate problems with "round tripping"[2] that occur when using concrete transport syntaxes based on RDF [2].

The OWL API attempts to present a highly reusable component for the construction of different applications such as editors, annotation tools and query agents. Besides allowing them to "talk the same language", it ensures that they share underlying assumptions about the way that information is presented and represented. Thus a cornerstone to the successful implementation and delivery of the Semantic Web, namely the *interoperability* of applications is achieved.

We draw inspiration from the impact that has been made by the provision of the XML Document Object Model (DOM) [20]. The DOM, along with freely available implementations (such as the Java implementations in Sun's JDK [18]) has allowed a large number of developers to use and manipulate XML in applications, which has in turn facilitated the widespread adoption of XML. Our hope is that a similar effect can be achieved with an API for OWL.

There is a long tradition for providing programmatic access to knowledge based systems, however most of the previous work has been centered around protocols, such as Open Knowledge Base Connectivity (OKBC) and Generic Frame Protocol (GFP), which are application programming interfaces for accessing knowledge bases stored in knowledge representation systems. Such protocol-centric approaches, automatically assume a client-server architecture for application development. However, our approach is rather component-based since our intention is to develop a reusable component for developing OWL-based applications, in style of DOM for XML-based applications. To our knowledge, there are no current existing implementations of APIs for the OWL language, however there have been previous related approaches.

DAML+OIL interfaces. There have been a number of similar initiatives to provide application interfaces aimed at precursors of OWL such as DAML+OIL[1]. Jena [9] supplies a DAML+OIL interface that provides convenience wrappers around their RDF interface in order to increase the efficiency of manipulating the DAML+OIL fragments embedded in a particular RDF file. Naturally, this approach gives a rather syntax-centric view of DAML+OIL. Additionally the implementation is bound to a particular RDF implementation. The DAML API by AT&T government solutions is an additional interface to DAML ontologies. It defines a structural interface for the manipulation and accessing of DAML ontologies that is not bound to a particular syntactic representation such as RDF.

Semantic applications. KAON [6] is an open-source ontology management infrastructure targeted for business applications. It includes a comprehensive tool suite allowing easy ontology creation and management, as well as the building of ontology-based applications. To the latter extent it defines a standard interface

[2] *Round tripping* refers to the process where a data structure (e.g. an ontology) is serialized and deserialized to/from some concrete syntax without loss of information.

to access semantic structures – the KAON API[3] – and multiple implementations there of, e.g. on top of relational databases. However, the ontology model supported in KAON is much less expressive than that described by OWL since an important focus of KAON is performance on large knowledge bases [13]. However, many of our underlying design considerations conceptually follow the KAON design.

Ontology Editors. OilEd [3] provided a collection of data structures representing DAML+OIL ontologies[4]. The OilEd data structures suffer in a number of ways however – some of the relevant issues are covered in other sections of this paper. One drawback is that the functionality is supplied as implementation *classes* rather than *interfaces*, which binds the client to a particular implementation of the model. In addition, support for tracking and recording change is minimal. Other ontology editors such as OntoEdit and Protege also expose their internal APIs to offer access to the underlying data structures but experience similar problems since their design is heavily influenced by the application purpose.

Ontology Versioning and Evolution. Since an API for manipulating ontologies has to address change in ontologies, previous work focused on this subject has been considered. [14] addresses change in DAML+OIL documents by providing diff-style comparison of individual documents and identification of changes by analysis of the differences, e.g. identifying the renaming of classes. [17] takes a different stance and identifies a change ontology, which captures the different types of changes that can occur in ontology modelling. The implementation within KAON encapsulates these different change types and allows the modification of changes via appropriate strategy objects (See Section 3.2), ensuring that change is carried out according to user specifications.

The remaining sections of the paper are structured as follows. Section 2 motivates some of the fundamental decisions taken in our design. Sections 3 and 4 discuss the design itself. Section 5 briefly describes examples of the use of the API, and we conclude with a summary of our contribution.

2 Separating Functionality

The OWL specification provides a description of the underlying language along with a formal semantics, giving a precise interpretation of the meaning of OWL documents or ontologies. What it means to be "an OWL Implementation" is, however, less clear. Indeed, an examination of the WebOnt Working Group [21] mail archives suggests that opinions differ widely as to what one can claim to be an implementation.

Different classes of application require, and provide, different aspects of functionality (See Figure 1). For example, a format/syntax translator acts as a client

[3] Available at `http://kaon.semanticweb.org`
[4] Available at `http://oiled.man.ac.uk`

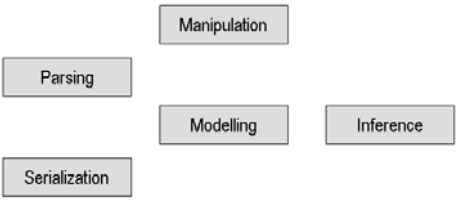

Fig. 1. Aspects of Implementation Support

```
A rdf:type rdfs:Class.
B rdf:type rdfs:Class.
B rdfs:subClassOf A.
b rdf:type B.
```

Fig. 2. Simple RDF Inference

of the API and requires the ability to parse, represent the results of the parsing in some way, and then serialize. An editing application would also require manipulation capabilities to allow construction and editing of ontologies (i.e. definitions of classes, properties and so on). A simple editor, however, need not actually require any functionality relating to semantics or inference, e.g. the facility for checking the consistency of class definitions, or whether subsumption relationships can be inferred. Alternatively, an application that simply deploys an ontology to client applications may not require any functionality that supports serialization, manipulation or extension of the ontology, but does support query of the ontology and its entailments. Turning to components that provide functionality, a reasoner will support inference, but need not be concerned with issues relating to serialization and parsing.

The following sections describe a number of examples that illustrate some of the issues we consider to be important. These include the need for explicit characterizations of functionality, the requirement for change support, identification of asserted and inferred information and preservation of ontological structure. These examples, along with the considerations above have motivated design decisions in our API as discussed in Section 3.

2.1 Entailment

Consider the RDF triples given in Figure 2. What might we expect when this collection of triples is given to an RDF-API and we then ask whether `b rdf:type A`? If the implementation simply represents the asserted facts as in the collection of triples, the answer is no. If, however, the implementation implements RDF entailment, then the answer is yes. It is not always clear in existing RDF implementations whether or not such entailments can be expected.

I)
```
Class(CarDriver partial
        Person
        restriction(drives someValuesFrom Vehicle))
```

II)
```
SubClassOf(CarDriver Person)
SubClassOf(CarDriver restriction(drives someValuesFrom Vehicle))
```

Fig. 3. Explicit Class Definition and Class Definition through Axioms

2.2 Explicit Change Operations

The ability to track change is important for a number of ontology-based applications. Editors must be able to record the actions that the user is performing if they are to be able to provide effective change management and versioning functionality. Similarly, clients of a central ontology service will need to be informed of updates and changes to the ontologies served by the server. Explicitly representing changes as first-class objects can support this (and more).

2.3 Information Grouping

Different application uses of OWL ontologies require different characteristics of the ontologies. For example, an application using an ontology in order to perform, say, search or indexing of information may only be interested in the underlying inferences that can be drawn from the axioms in the model. An editing application, or one that provides a graphical view on the ontology in order to support query, may have different requirements in that the application may need to know the way in which the information has been structured or grouped. A particular short-coming identified in DAML+OIL [2] was the inability to distinguish the way that information had been presented by the original modeller or ontologist. The OWL abstract syntax, however, allows the definition of classes using both a definitional style, i.e. the use of class definitions, and through general axioms, i.e. the use of axioms.

As an example, consider Figure 3: I) shows the use of a class definition. In this case, the class of `CarDrivers` is defined as a subclass of the intersection of `Person` and those things that `drive` a `Vehicle`. This definition could also be made through a pair of subclass axioms as in Figure 3 II). Both definitions have the same *semantic* effect (in terms of the underlying model), but we can argue that these are, in fact, different. The way in which the information is presented is part of the OWL ontology, and an API for the language should try and preserve this wherever possible.

As a more complicated example, consider the three alternative definitions shown in Figure 4. Again, all three of these provide exactly the same semantics in terms of the inferences that can be drawn. However, they convey slightly different ways of modelling the world in terms of how the ontologist thinks things fit together. As discussed in [3], the issue here is that we would like to ensure that not only do we capture the correct semantics of the ontology, but also the *semiotics* [7].

I)
```
Class(CarDriver complete
        Person
        restriction(drives someValuesFrom Car))
SubClassOf(CarDriver PersonOver17)
```

II)
```
Class(CarDriver partial
        PersonOver17)
EquivalentClasses(CarDriver
        intersectionOf(Person
                        restriction(drives someValuesFrom Car)))
```

III)
```
SubClassOf(CarDriver Person)
SubClassOf(CarDriver restriction(drives someValuesFrom Car))
SubClassOf(CarDriver PersonOver17)
SubClassOf(intersectionOf(Person
                          restriction(drives someValuesFrom Car))
           CarDriver)
```

Fig. 4. Alternative Class Definitions

The ability to preserve these distinctions within the API is an important one, particularly if the API is to support not only the deployment of ontologies to applications but also applications that bring the user closer to the actual ontology, such as editors.

2.4 Assertions and Inferences

We consider that a separation of assertion and inference is important for applications such as editors. To illustrate this, we draw on our experiences with the implementation of OilEd [3].

OilEd used a DL reasoner to compute the inferred subsumption hierarchy of a DAML+OIL model[3]. There are a number of scenarios where this can prove useful. For example, one use case is the enhancement of RDF Schemas. The Schema is read into the tool, and the increased expressivity of DAML+OIL can be used to provide more detailed descriptions of the classes (for example the definition of `CarDriver` as a person who drives a vehicle in Figure 3). Once the descriptions have been applied, we can then export the schema in RDF(S) again. The original language (RDF(S)) is not rich enough to represent many of the constructs available in DAML+OIL, so these class definitions will be lost in the resulting output. Before exporting, however, we can use the reasoner to compute the inferred hierarchy (which may well include new subclass relationships due to the assertions), and then serialize the schema with the additional relationships. In this way the inferred sub/superclasses can be made accessible to simple RDF applications.

During the development of the tool, users expressed a desire to have the ability to add this inferred information back into the ontology. Thus a “commit

```
Class(Vehicle)
Class(Car partial Vehicle)
Class(Person)
ObjectProperty(drives)
Class(Driver complete
 intersectionOf(
  Person
  restriction(drives
    someValuesFrom Vehicle)))
Class(CarDriver complete
 intersectionOf(
  Person
  restriction(drives
    someValuesFrom Car)))
```

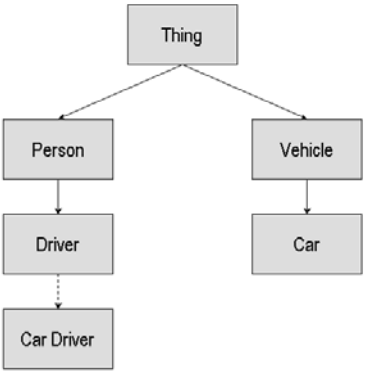

Fig. 5. A Simple Ontology and Inferred Hierarchy

changes" button was added, which did precisely this. In OilEd's implementation, this was achieved by adding the information to the assertions which make up the model.

Although the addition of the inferred relationships does not change the underlying semantics of the ontology (as they are already inferred, we are simply adding redundant information), over time our experience was that this was a confusing process, in particular when users then wanted to further edit the amended ontology. For example, take the simple ontology[5] shown on the left of Figure 5. This produces a hierarchy as shown on the right of Figure 5

In this example we find an inferred subclass relationship between `CarDriver` and `Driver` (shown as a dotted line). A simple approach would be to add this relationship back into the ontology. Consider the situation now, however, where the user is presented with the concept hierarchy in a Graphical User Interface (GUI), and tries to use the hierarchy to directly manipulate the underlying ontology, for example removing the relationship between `CarDriver` and `Driver`. How should we interpret this within the application? In order to *truly* remove the relationship between the two classes, we would need to alter their definitions, rather than simply removing some sub/superclass link between them. In this example, the user could remove the sub/superclass relationship and then find that it "comes back" after a reclassification.

The key issue here is that the information regarding the class hierarchy can be considered as *inferred* information which can be calculated from the *asserted* information which is present in the axioms of the ontology. We consider it to be of benefit to explicitly represent this split in the API.

In this way, in our example, the user interface can inform the user that more action than simply removing the super/subclass link is required.

[5] Our examples use the OWL Abstract Syntax[15] for presentation of ontology fragments

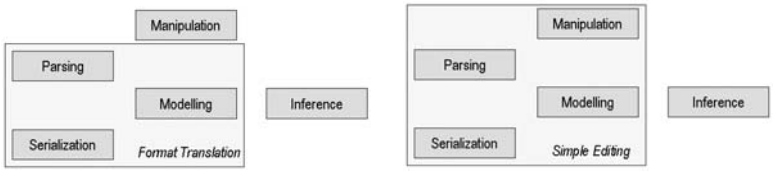

Fig. 6. Aspects and Applications

2.5 Aspects of Functionality

In the light of the preceding discussion, we can consider a number of different tasks that applications may perform which could be thought of as providing "OWL implementation". These include:

Serializing. Producing OWL concrete syntax (for example as RDF triples or using the OWL presentation syntax) from some internal data structure or representation;

Modelling. Providing data structures that can represent/encode OWL documents. This representation should be at an appropriate level. An XML string would provide a representation of the information in an ontology, but is unlikely to facilitate access to that information;

Parsing. Taking a concrete representation of an OWL document (e.g. an RDF-XML serialization of an OWL document) and building some internal representation that corresponds to that document;

Manipulation. Providing representation along with mechanisms for manipulation of those documents;

Inference. Providing a representation that in addition implements the formal semantics of the language.

We can think of these different tasks as providing different aspects of support for OWL (See Figure 1). Some aspects will (in general) require support from others, although this is not entirely the case. For example, serialization can be seen as a minimal level of support that does not necessarily require the implementation to "understand" or represent the entire language.

As introduced above, different classes of application will need differing combinations of these classes of functionality, as illustrated in Figure 6. We see this separation of the classes of functionality an application provides as crucial if we are to be confident that the implementation supplies appropriate functionality. Our API design explicitly reflects this through the separation of functionality into distinct packages.

3 API Design

The API contains a number of different packages, each of which reflects an aspect of functionality as introduced above.

3.1 Model

The `model` package provides basic, read-only access to an OWL ontology. Thus there are methods for accessing the Classes defined or used in the ontology (and their definitions), the Properties defined or used, Axioms asserted and so on.

The data structures and accessor methods defined within this package reflect the requirements expressed in Section 2.3 for the explicit preservation of information grouping. Although this introduces a certain amount of redundancy into the data structures (as there are multiple ways of representing information) it allows us to ensure that no information loss occurs when representing ontologies using the API.

For the situations where applications are not concerned with the grouping or structuring of information, we can provide alternative "views" of the information in the ontology, e.g. an axiom-centric view that simply presents all the assertions relating to class definitions as subclass axioms. This can be achieved through the use of `helper` classes.

3.2 Change

The `model` package described above provides read-only access to ontologies. The `change` package extends this to allow manipulation of those structures, e.g. the addition and removal of entities, changes to definitions, axioms and so on.

The `change` package achieves this through the use of the **Command** design pattern [8] which encapsulates a change request as an object. Changes are then enacted by a ChangeVisitor. See Section 4.2 for further discussion.

3.3 Inference

The OWL specification includes a detailed description of the semantics of the language. In particular, this defines precisely what entailment means with respect to OWL ontologies, and provides formal descriptions of properties such as consistency. The implementation of these semantics is a non-trivial matter, however, and providing a *complete* OWL reasoner, effectively requires the implementation of a Description Logic (DL) theorem prover. By separating this functionality, we can relieve implementors of the burden of this, while allowing those who do provide such implementations to be explicit about this in their advertised functionality. The `inference` package is intended to encapsulate this and provide access to functionality relating to the process of *reasoning* with OWL ontologies.

In addition, the `inference` and `model` packages partition functionality along the lines described in Section 2.4 above. This does not completely solve all the associated problems of supporting user editing of the ontology via graphical means, but by exposing the particular kinds of information that are present in the ontology, we are making it clear to applications what they can, and can not, do.

Of course, providing method signatures does not go all the way to advertising the functionality of an implementation – there is no guarantee that a component implementing the `inference` interface necessarily implements the semantics correctly. However, signatures go some way towards providing an expectation of the operations that are being supported. Collections of test data (such as the OWL Test Cases [4]) can allow systematic testing and a level of confidence as to whether the implementation is, in fact, performing correctly.

4 Detailed Design Decisions

The following sections discuss our design decisions in more detail.

4.1 Modelling the Language

Syntax vs. data model. The API represents the OWL language by modelling the language constructs in a data model. Often, such a data model might closely reflect the syntax of the language. Since OWL has several so-called presentation syntaxes, however (XML and RDF for the time being), the syntactic constructs available in the language cannot be used as the basis for establishing the data model. Adopting a bias towards a particular presentation syntax, e.g. the RDF representation, imposes major difficulties for access since it involves many syntactic overspecifications that are due to the particularities of the data model. For example, n-ary language constructs such as intersection and union, are broken down into several triples in the RDF graph. It is easier to access and manipulate these constructs, if they are presented as n-ary ones. Since any presentation syntax relates to the abstract syntax of the language [15], our decision is that the data model should follow this abstract syntax. In consequence, the mapping between serializations and data model is carried out by the parser and serialization implementations.

Interface vs. Implementation. The data model itself is represented as an interface, allowing user applications to provide alternative implementations of the interfaces with different properties. The use of interfaces is, of course, standard practice in Object Oriented design. It is however worth mentioning here, though, as it ensures that client applications can use the API without being concerned about the particular implementation strategy. Thus an implementation could provide simple in-memory storage of the ontologies (as is the case with our draft implementation), or could provide some persistent storage mechanism, with the interface sitting on top of a relational database, or an RDF store.

The data model is represented as an extensive interface hierarchy (see Figure 7 for an overview). This allows the simplification of a possible implementation by reusing abstract implementations for similar behaviour. For example, an implementation of functionality for traversing the property hierarchy can be used for both datatype properties and object properties.

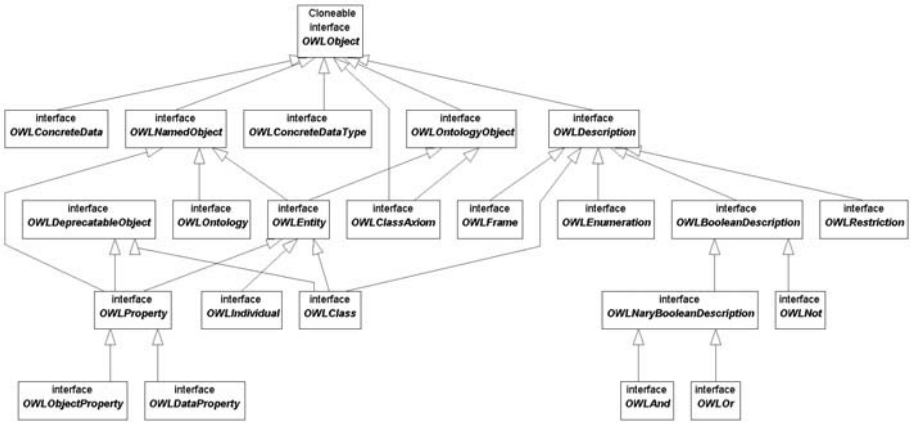

Fig. 7. OWL Data Model Excerpt

Locality of Information. All assertions are associated with a particular ontology, and OWL allows different ontologies to make different assertions about the same classes and properties. In order to support this, we require the ability to distinguish the source of information. The methods specified in interfaces maintain such information. For example, iterators for the declared superclasses of a given class can take an ontology as an argument, which restricts the iteration to those declarations made within the context of the given ontology.

4.2 Change

A critical point in applying ontologies to real-world problems is that domains are dynamic, and change over time – new concepts evolve, concepts change their meaning etc. Thus, the support for change is a crucial feature in an OWL API. Change support has to meet several aspects.

Granularity of change. Change in ontologies occurs at differing granularities. Besides basic changes, such as adding and removing entities, (classes, class restrictions or properties) change also happens at a higher granularity. For example, a user may decide to create a new class `Vehicle` that subsumes existing classes such as `Bike` and `Car`. A user may achieve this through successive application of fine grained changes. In the above example, 3 operations might be required: adding `Vehicle`, adding two superclass axioms and additional changes to keep the class hierarchy consistent, e.g. moving common existing superclasses of `Bike` and `Car` to `Vehicle`.

However, it can be beneficial to capture the high level intention of the above changes in a composite change operation tailored for this purpose. The impedance mismatch between the intention of change and its achievement is then removed, and the possibility of conceptual errors is decreased. Table 1 presents some composite changes supported by the API.

Table 1. Some composite changes (following [17])

Composite change	Description
Merge classes	Replace several classes with one aggregating their instances
Extract subclasses	Split a class into several classes and distribute properties among them
Extract superclass	Create a common superclass for a set of unrelated classes and transfer common properties to it
Pull up properties	Move property domains from a class to its super class

Dependency of Change. As we can see from the above example changes are not isolated – on the contrary most basic changes are performed in response to other basic changes. This creates a natural chain of changes. The API supports this by allowing the representation of chains of changes. This information often proves useful if a given change should be undone at a later stage, since it indicates the context within which the change was carried out. Composite changes are automatically decomposed into basic changes in the implementation and chained appropriately.

User intention. The above information is not always sufficient to capture the intention of a change completely, due to its incomplete specification. Hence, users are able to specify different change strategies. The choice made for a given strategy allows customization of the way that changes are processed depending on the particular situation and strategy.

For example, a user may choose to compute additional changes to keep a consistent structure of the ontology. For example, when deleting a class all instances may be chosen to be deleted as well, or to be moved to other classes. The particular choice here will be application, task, or context specific.

Change strategies. Change strategies can be used to support various aspects of customizable implementation behaviour. For example, a problem during the evolution of a Description Logic ontology is implicit meaning change in classes. Changes in axioms or definitions may effect the inferences that can be drawn from an ontology, implicitly impacting the "meaning" of classes. Reasoning may come into play during the enactment of evolution strategies in order to control this. For example, an implementation may choose to prevent any changes that cause inconsistencies to occur within the ontology.

Design decisions. The API acknowledges the above issues by separating the representation of change from the processing of changes. Change representation, e.g. the addition and removal of entities, changes to definitions, axioms and so on, is achieved through the use of the **Command** design pattern [8] which encapsulates a change type as a class. While the API "ships" with a complete set of basic change commands for all elements of the language, it also provides an elementary set of composite change commands such as represented in Table 1.

Users can provide their own change commands by subtyping an existing change class, and extending the processing of changes accordingly.

The processing of changes are then enacted by a ChangeVisitor. This approach has also been used with success in the KAON architecture and API. Along with the use of the command pattern, we can use the **Strategy** pattern [8] and employ customizable change strategies in a ChangeVisitor, which can edit or manipulate streams of change events to ensure that the internal models are kept in consistent states. For example, a particular implementation of the OWL Ontology interface may expect that before any axioms involving a class can be added to the ontology, the ontology must contain a (possibly empty) definition of the class. This is not necessarily something that we wish to be true of **all** implementations. However, we want to preserve the possibility to do so in a particular implementation, which can employ an appropriate strategy that takes any such axiom additions and first ensure that the classes used in the axiom are added to the ontology, thus preserving the internal consistency of the data structures.

Advanced Features. The use of the command pattern facilitates support for operations such as undo or redo, and the encapsulation of changes as operations provides a mechanism with which to track changes and support version management. The change objects also provide a convenient place for storing metadata about the changes, for example the user who requested the change – information which is again crucial in supporting the ontology management and editing process. In future versions of the API, changes may be encapsulated in transactions, which are processed as such, meeting the basic properties of transactions in databases, i.e. Atomicity, Consistency, Isolation and Durability.

4.3 Parsing

Since OWL possesses several presentation syntaxes, parsing is decoupled from particular implementations of the data model. A given file or stream is consumed by parser components, which issue a sequence of change events to the API in order to build an in-memory representation of an ontology.

Parsing RDF. Parsing RDF is a non-trivial effort. If possible, parsing should be done in a streaming manner to avoid large memory consumption while parsing large ontologies. However, with RDF this is, in the general case, impossible, since the graph is not serialized in any particular order. We *cannot* guarantee that all information required to process a particular syntactic construct is available until the entire model has been parsed.

Additionally, RDF ontologies are, in our experience, prone to errors due to their dependency on URIs. In many DAML+OIL ontologies inconsistent use of URIs was made. For example, namespaces are often misspelled. Hence, a series of heuristics are required in practice that try to ameliorate user errors by implicitly correcting such misspellings.

Another source of difficulty for parsing are missing definitions of classes or properties used within the ontology. For example, in the case of OWL Lite and OWL DL, properties must be explicitly typed as object properties or datatype properties. We can not tell from the URI what the correct type of the property, i.e. datatype or object property, is. Instead, a series of heuristics must be applied, e.g. inspecting all property instantiations and deducing from usage what the correct type could be.

However, the applied parsing heuristics must be optional and their usage must be specified by user applications. For example, a species validator (see Section 5) does not want to receive a cleaned ontology, since it could not detect the correct species of the initial source if definitions have been tampered with.

Inclusion The OWL language provides a simple mechanism for inclusion and import. Inclusion in the API is dealt with by registering all open ontologies within a housekeeping facility in the API. This facility manages all available parsers, thereby allowing the inclusion of XML-based ontologies into RDF-based ontologies, and manages the formal dependencies between open ontologies. This avoids the re-parsing of multiply used ontologies.

The parsing of included ontologies is handled in a depth-first manner. However, in case of RDF, the complete parsing of a certain RDF model is required, since the triple stating an inclusion could be the very last triple parsed in the model. The necessity to keep all RDF models in an inclusion hierarchy in-memory requires large amounts of main memory. This raises question about the suitability of RDF for large ontologies, which can be processed with low memory footprint in the XML-based syntax.

Inclusion is another fragile component of OWL due to the dependency on URIs. For example, a draft version of the OWL Guide `wine` ontology [16] contained an incorrect `imports` reference to the OWL Guide `food` ontology. To lessen this problem we adopt a solution in the OWL API of distinguishing between logical and physical URIs. The logical URIs are the base names for most URIs in the ontology, while the physical URIs refer to the actual locations that ontologies can be retrieved from.

The base name can be set in an explicit serialization via the `xml:base` attribute. This helps to enforce good practise with respect to relative URIs, since they do not then change if the ontology physically moves. If `xml:base` is not used, relative URIs are resolved relatively against the physical file URI.

The implementation keeps track of both logical URIS and physical URIS and can locate ontologies by either URI. In an extension of the OWL API, we could provide a further means for locating ontologies, by means of a registry as described in [10]. In the example above, if the implementation is unable to find the ontology by physical URI, it can try to locate instead by logical URI.

4.4 Implementation Language

The use of Java introduces a number of limitations on the API. For example, without generic collections, it is difficult to guarantee type safety without

introducing a large number of extra helper classes to represent, for example, collections of Classes or collections of Properties. In our design, we have chosen simplicity over type safety, and a large number of methods simply return `Sets` when collections of objects are expected. It is then up to client applications to cast to the appropriate objects. However, this situation is remedied with the upcoming version of Java which supports generic collections.

5 Example: Species Validation

Finally, we present an example application that has been built using our draft implementation[6]. Species Validation[4] is the process whereby we identify the particular OWL sub-species (Lite, DL or Full) that an ontology belongs to. Species identification requires two stages:

1. parsing the OWL document;
2. a post-process to ensure that the various conditions for membership of the species hold. Examples of the validation conditions are: OWL Lite ontologies should not contain `unionOf`, `complementOf` or `oneOf` expressions; OWL DL/Lite ontologies cannot include properties specified as transitive with a super property specified as functional; OWL DL/Lite ontologies must separate Classes, Properties and Individuals, and cannot make use of metamodelling devices such as "Classes as instances".

Parsing requires access to the `model` and `change` interfaces, while the post processing stage is simply read-only and thus only uses the `model` interface. We have implemented a simple OWL Validator that performs validation and which is accessible as a Java Servlet.

The validator has been tested using the OWL Test Suite [4] and a draft version of the wines ontology from the OWL Guide [16]. In the wines ontology, a number of minor errors were detected. These included misuse of vocabulary (e.g. `hasClass` rather than `someValuesFrom`) and inconsistency in capitalization. In the latter case, the miscapitalization leads to the ontologies being flagged as outside the DL subspecies as the resulting IndividualIDs are not then explicitly typed (one of the conditions for OWL DL). The tool was also able to identify a number of minor errors in proposed tests. The validator was also able to read, check and detect errors in a draft version of a large ontology[7] containing some 500,000 RDF triples.

6 Conclusion

As a result of our work the OWL API is a readily available Standard Application Programming Interface (API)[8] that allows developers to access data structures

[6] See `http://wonderweb.semanticweb.org/owl/` for updates on the API development, current documentation, and links to applications using the API.

[7] The National Cancer Institute Thesaurus `http://www.mindswap.org/2003/CancerOntology/nciOncology.owl`

[8] It is available at `http://wonderweb.man.ac.uk/owl/`

and functionality that implement the concepts and components needed to build the Semantic Web. The higher-level abstractions of the API help to insulate application developers from underlying issues of syntax[9] and presentation.

The OWL Test Cases [4] provide general notions of OWL syntax and consistency checkers, but this is a somewhat coarse-grained idea – an OWL consistency checker takes a document as input, and outputs one word being `Consistent`, `Inconsistent`, or `Unknown`. Real applications need a finer notion of what is being implemented along with richer descriptions of functionality. Our design facilitates this through the explicit characterization of different aspects of functionality.

An exemplar of this approach is the XML Document Object Model (DOM) [20]. The DOM provides "a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents." In practice, implementations of the DOM, such as the Java implementations encapsulated in the `org.w3c.dom` packages included in the latest releases of Sun's Java Software Development Kit (SDK) [18] (along with associated parsing libraries) have allowed a large number of developers to use and manipulate XML in applications. Similarly, APIs for the Resource Description Framework (RDF) [5] such as Jena [9] and the Stanford RDF API [12] have helped to push deployment of RDF technology into applications.

Our hope is that the OWL API will become a predominant component in the Semantic Web application world and gain a similar status as the standard SAX and DOM [20] interfaces for XML (or at least serve as a starting point for discussion about the design of such infrastructure).

Acknowledgments. The authors would like to thank Angus Roberts, who contributed to the early design of a prototype API. This work was supported by the European FET project WonderWeb (EU IST-2001-33052) and the myGrid E-science pilot (EPSRC GR/R67743). Raphael Volz was supported by DAAD.

References

1. Joint US/EU ad hoc Agent Markup Language Committee. Web Ontology Language, Reference Version 1.0. `http://www.daml.org/2001/03/daml+oil-index.html`.
2. S. Bechhofer, C. Goble, and I. Horrocks. DAML+OIL is not enough. In *SWWS-1, Semantic Web working symposium*, Jul/Aug 2001.
3. S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. OilEd: a Reason-able Ontology Editor for the Semantic Web. In *Proc. of KI2001, Joint German/Austrian conference on Artificial Intelligence*, volume 2174 of *LNAI*, pages 396–408, Vienna, Sep 2001. Springer-Verlage.
4. J. Carroll and J. De Roo. Web Ontology Language (OWL) Test Cases. `http://www.w3.org/TR/owl-test/`, 2003.

[9] **Quiz Answer:** `type` and `Property` are in the RDF vocabulary, `comment` is in RDF(S). Bonus point if you got all three.

5. World Wide Web Consortium. Resource Description Framework (RDF). http://www.w3.org/RDF/.
6. E. Bozsak et.al. KAON – Towards a Large Scale Semantic Web. In K. Bauknecht, A. Min Tjoa, and G. Quirchmayr, editors, *EC-Web 2002*, volume 2455 of *Lecture Notes in Computer Science*, pages 304–313. Springer, September 2002.
7. J. Euzenat. Towards formal knowledge intelligibility at the semiotic level. In *Proc. of ECAI 2000 Workshop Applied Semiotics: Control Problems*, pages 59–61, 2000.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
9. Hewlett Packard. Jena Semantic Web Toolkit. http://www.hpl.hp.com/semweb/jena.htm.
10. A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. An infrastructure for searching, reusing and evolving distributed ontologies. In *To Appear In Proc. of WWW 2003*, 2003.
11. R. Meersman and Z. Tari et. al, editors. *Proc. of the International Conference on Ontologies, Databases and Applications of SEmantics ODBASE 2002*, volume 2519 of *LNCS*, University of California, Irvine, USA, 2002. Springer.
12. S. Melnik. Stanford RDF API. http://www-db.stanford.edu/~melnik/rdf/api.html.
13. B. Motik, A. Maedche, and R. Volz. A conceptual modeling approach for building semantics-driven enterprise applications. In Meersman and et. al [11].
14. N. Noy and M. Klein. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 5, 2003.
15. P. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language (OWL) Abstract Syntax and Semantics. http://www.w3.org/TR/owl-semantics/, 2003.
16. M. Smith, C. Welty, and D. McGuinness. OWL Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/, 2003.
17. L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. In *Proc. of EKAW 2002*, page 285 ff., 2002.
18. Sun Microsystems, Inc. Java™Platform. http://java.sun.com/j2se/.
19. Y. Sure, S. Staab, and J. Angele. OntoEdit: Guiding ontology development by methodology and inferencing. In Meersman and et. al [11].
20. W3C DOM Working Group. Document Object Model. http://www.w3.org/DOM/.
21. W3C WebOnt Working Group. http://www.w3.org/2001/sw/WebOnt.