Applied Ontology 7 (2012) 113–132
DOI 10.3233/AO-2012-0107
IOS Press


# Engineering use cases for modular development of ontologies in OWL

Alan Rector *, Sebastian Brandt, Nick Drummond, Matthew Horridge, Colin Pulestin and Robert Stevens
*School of Computer Science, University of Manchester, Manchester, UK*

**Abstract.** This paper presents use cases for modular development of ontologies using the OWL imports mechanism. Many of the methods are inspired by work in modular development in software engineering. The approach is aimed at developers of large ontologies covering multiple subdomains that make use of OWL reasoners for inference. Such ontologies are common in biomedical sciences, but nothing in the paper is specific to biomedicine. There are four groups of use cases: (i) organisation and factoring of ontologies; (ii) maintaining stable interfaces and bindings between ontologies and between ontologies and software; (iii) localization of ontologies to the requirements of specific sites and (iv) extension of ontologies and encapsulation of modifications. OWL's axiom-oriented import mechanism has many similarities with import mechanisms in object-oriented software but also important differences – in particular, the effects of OWL imports are global, and the order in which modules are imported is irrelevant. The advantages and disadvantages of OWL's axiom-oriented approach are discussed, and suggestions are made for extensions to allow axioms to be filtered out as well as added – a mechanism that we term "adaptation" to distinguish it from the standard import mechanism. Finally we discuss possible alternatives and practical experience with the approaches presented.

Keywords: Ontologies, modules, API, development methodology, OWL

## 1. Introduction

Ontology modularization covers two separate topics:

- *Module extraction*: Decomposing existing ontologies into separable modules using various logical criteria, e.g., Grau et al. (2008), Sattler et al. (2009), Del Vescovo et al. (2010), Seidenberg and Rector (2007).
- *Modular development*: Development of ontologies in modules by analogy to modular software engineering methodologies, to promote re-use, collaborative development and clean design, e.g., Rector et al. (2008), Bao et al. (2006), Pathak et al. (2009), Thomas et al. (2006).

This paper is concerned with the second topic: modular development. To the best of our knowledge, the closest similar work is that by Ensan and Du (2008), although they approach the issue from a formal perspective rather than focusing specifically on use cases.

Our work focuses on engineering large ontologies in OWL profiles that support efficient reasoning, usually OWL-DL or OWL-EL (OWL, 2009). The ontologies we work with range in size from 5000 to 500,000 classes, and usually cover multiple sub-domains. Typically they have no or minimal A-boxes. Such ontologies are common in biomedical applications which is the authors' area of research,

*Corresponding author: Alan Rector, School of Computer Science, University of Manchester, Manchester M13 9PL, UK. E-mail: rector@cs.manchester.ac.uk.

1570-5838/12/$27.50 © 2012 – IOS Press and the authors. All rights reserved

e.g., the various OBO-Foundry ontologies (Smith, 2007), SNOMED-CT (IHTSDO, 2011; Stearns et al., 2001), or the Foundational Model of Anatomy (Rosse et al., 1998). However, although the examples in this paper are from biomedicine, large ontologies increasingly appear elsewhere, and nothing in the approaches to modularization described in this paper is specific to biomedicine.

The methods presented exploit the OWL imports mechanism and assume that modules will be maintained in separate files. While OWL provides other mechanisms that might be used to distinguish modules – e.g., by the use of IRIs and/or namespaces – using separate files for modules along with the OWL imports mechanism is the most suitable for large ontologies. Moreover, our primary concern here is with the use cases; we leave it to others to show how alternative mechanisms might address the same use cases.

The "modular development" methods presented here assume that entire modules ("ontologies" in the OWL sense) will be imported, although those ontologies may themselves be "extracted modules" from some larger ontology. However, the detail of constructing appropriate module extraction mechanisms for this purpose is outside the scope of this paper.

The OWL imports mechanism has many analogies with software import mechanisms, but also important differences. When one OWL module imports another, the axioms are simply aggregated. Hence, order of importing is irrelevant; multiple routes to an imported module, and mutual bi-directional imports are unproblematic. However, unlike most software systems, in which classes and methods are local to a module, OWL axioms and entities are always global and can affect any entity in either importing or imported modules. Hence, without conventions to manage their scope, OWL modules have the potential to interact in ways that are neither expected nor intended.

Furthermore, the OWL imports mechanism is low level and unstructured. For example, OWL has no built-in notion of an "Interface" analogous to the software notion of an Application Programming Interface (API).

To explore the process of developing ontologies using modules, we discuss four main use cases, two with variants:

*Use case* 1: Ontology organisation and factoring.

1.1 Simple import of upper and annotation modules.
1.2 Normalization of ontologies.

*Use case* 2: Interfaces between ontologies and between ontologies and software.

2.1 Between ontologies.
2.2 Between ontologies and software.

*Use case* 3: Ontology localisation.
*Use case* 4: Ontology extension.

In the light of these use cases, we discuss the advantages and disadvantages of the OWL axiom-oriented approach to imports by comparison to the more familiar object-oriented approach and argue in favour of an extension of the notion of "imports" to "adaptation" to improve the ability of modules to encapsulate changes and support more flexible localization.

Finally, two notes. There is a problem in vocabulary. The OWL standard (Patel-Schneider et al., 2004) makes no mention of "modules". It speaks only of one "ontology" importing another. However, we find it easier to refer to each individual "ontology", in the standard's sense, that is imported or imports another as a "module", and to the entire ensemble as an "ontology".

Also, although we use human readable labels in figures and axioms throughout this paper to improve readability, the underlying operations are always between the identifying IRIs. The labels are incidental to the logic. (It is almost universal practice in the biomedical community to use “nonsemantic” identifiers – in OWL usually IRIs with meaningless numeric endings – plus label annotations for the human-readable names. Use of text or other “semantic identifiers” in ontologies intended to be shared, sustained, or used across language communities has been found to be unmaintainable and has long been deprecated in the biomedical community, as codified in Cimino’s seminal paper (Cimino, 1998).)

## 2. Use cases

In the examples that follow, we shall use a simplified idealization of work on the Ontology of Clinical Research, OCRe (Sim et al., 2010) augmented by work on ontology based software in a large collaborative project on health informatics (Pulestin et al., 2008a).

The goal of the Ontology of Clinical Research (OCRe) is to describe the design of clinical research studies. It needs a vocabulary of diseases and outcomes and the relations between them, but the Ontology of Clinical Research itself does not concern itself with specific diseases, indeed it wants to be applicable to studies of different disease groups that are, in fact, represented in different ontologies – some in SNOMED (College of American Pathologists, 2010), some in the National Cancer Institute Thesaurus (de Coronado et al., 2004), and some in more specialised subspecialty ontologies. Likewise, it is not itself interested in maintaining a catalog of drugs and surgical procedures – collectively referred to here as “interventions” – but needs to draw on external resources, some public, and some private. The Ontology of Clinical Research itself needs to know only that diseases and interventions exist, the roles they play in studies, and a few relations between them pertinent to the design of research studies.

We summarise the modules as simplified and used in this paper in Table 1. We shall refer to the annotation modules and upper ontology modules together as “generic modules” or “generic ontologies” and the others as “domain modules” or “domain ontologies”.

Table 1

Summary of modules used in examples in this paper

| Name in this paper | Description and comment |
|---|---|
| Annotation | Editorial and provenance information, rights, comments, labels, etc., e.g., Dublin Core, skos:prefLabel, etc. |
| Upper ontology | The most generic entities such as physical object, process, role, etc., e.g., BFO, DOLCE, BioTop, etc. |
| Anatomy | Normal and variant structures of the body |
| Pathology | What goes wrong in disease (“morphology” in SNOMED CT parlance) |
| Disorders | The things that go wrong with the body |
| Diseases | The entities usually described as diseases, for purposes of example entailing a disorder and cause |
| Causes | Entities that can cause disorders but do not fall in any of the other modules, e.g., micro-organisms, toxins, sources of injury, etc. |
| Drugs | Substances used as drugs and the products based on them |
| Surgical procedures | All forms of procedures performed on patients |
| Interventions | Drugs, surgical procedures, and other treatments. |
| Clinical research | Structure of research studies, analyses, experimental and control groups, etc. The Ontology of Clinical Research itself. |

Although the Ontology of Clinical Research explicitly wishes to be neutral as to diseases, interventions, etc., users of the Ontology of Clinical Research for cataloguing specific sets of studies need to be able to formulate queries that involve notions both from the ontology itself and from the contributing modules. A typical query might need entities from several or even all of the modules listed in Table 1, e.g., find: all *randomly controlled trials* (clinical research) of *Stage 2c Breast Cancer* (disease) *treated with* (intervention) *Local excision* (intervention) *followed by* (clinical research) *Tamoxifen* (intervention). Hence, users want to import a combined ontology that expresses diseases and interventions (and many other factors) in ways that can be used in conjunction with the Ontology of Clinical Research. Furthermore, software needs to be written that can deal with any research study using the Ontology of Clinical Research in conjunction with any of the various disease and intervention ontologies needed for different kinds of studies. Therefore, an overall strategy is to import all modules to form an integrated ontology and then use it in software is required.

### 2.1. Organisation and factoring

#### 2.1.1. Simple import of upper and reference ontologies

The simplest use of OWL's import mechanism is to import common standard reference modules such as the Dublin Core (Weibel et al., 1998) for annotation or SKOS (Isaac & Summers, 2009) for labelling. In addition, many collaborating groups use a common upper ontology. For example, almost all OBO-Foundry ontologies (Smith, 2007), which include the widely used Gene Ontology, begin by importing BFO 1.1 and the Relation Ontology, each of which imports the Dublin Core ontology. As another example, most of the ontologies developed under the BootStrep (Bootstrep Consortium, 2011), Debugit (Debugit Consortium, 2011) and Aneurist (Aneurist Consortium, 2011) EU programmes import BioTop (Schulz et al., 2006) as a common upper ontology.

An abstraction of this use case is shown in Fig. 1, in which an upper ontology module imports an annotation module and is in turn imported by domain modules.

#### 2.1.2. Normalization and bridging

In a previous paper (Rector, 2003), some of the authors introduced the notion of a "normalized" ontology, as a means of ensuring clean design and clean separation of entities. That paper contains a full discussion of normalization. However, a brief summary is given here.

An ontology may be described as "normalized" if and only if:

(1) The named classes can be partitioned into (a) "primitive" classes that appear on the left-hand-side only of subclass axioms and (b) "defined" classes that appear on the left-hand-side of at least one equivalent-class axiom.
(2) Every primitive class is a subclass of exactly one other primitive class.
(3) All the primitive direct subclasses of any primitive class are disjoint.

The result is to partition the primitive classes of the ontology into disjoint trees or "taxonomies". Typically, each primitive taxonomy represents a different subdomain, e.g., anatomy, substance, function, role, etc. This allows the descriptors that belong to each subdomain itself to be distinguished from those that arise because of the interactions between subdomains. For example, in the pathology module, *Carcinoma* is defined as a specific kind of malignancy with various stages. In the anatomy module, *Breast* is defined as an organ containing ducts, secreting milk, etc. In the interventions ontology *Tamoxifen* might be defined as a drug that had a particular chemical structure, suppresses certain hormones, etc.

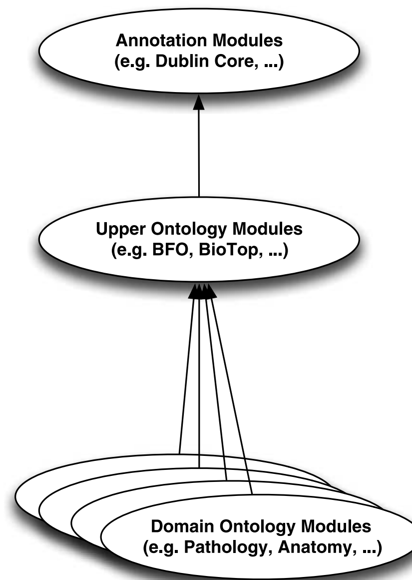


Fig. 1. Simple import of annotation and upper ontologies.

Axioms involving classes from different primitive hierarchies are known as “bridging axioms”[1] because they “bridge” between notions from different subdomains. Bridging axioms are normally placed in separate “bridging modules”.

In the original paper (Rector, 2003), both the primitive taxonomies and the bridging axioms were implemented in a single ontology in a single file, because, at the time of publication, tools for modular development were cumbersome. Tooling in most modern ontology development systems overcomes this barrier, and makes it easy to develop normalized ontologies as sets of separate modules. (We use Protégé 4.1 (Horridge et al., 2004), but most other toolsets – e.g., NeOn (Hasse et al., 2008, NeOn Consortium, 2011), TopBraid composer (TopQuadrant, 2011) – also provide convenient support for imports.)

We now routinely implement each major tree in a separate “subdomain module” and then provide one or more “bridging modules” that contain the axioms and definitions that link the subdomain modules together. As in Fig. 2, this process often cascades, so that what are bridging modules at one level become subdomain modules at the next, e.g., the *Anatomy* and *Pathology* subdomain modules are bridged by the *Disorders* module, which in turn becomes one of the subdomain modules along with *Causes* that is bridged by the *Disease* module, which itself is bridged with the *Drugs* and *Surgical Procedures* modules by the *Interventions* module.

An example of a bridging axiom between the *Anatomy* and *Pathology* modules might be the definition of *Carcinoma of the breast* (the most common form of breast cancer) as:

Carcinoma_of_breast *EquivalentTo: Carcinoma that* has_locus *some* Breast

[1]In the original paper, the phrase “binding axiom” is used, but we reserve that phrase in this paper for those axioms that implement Ontology Binding Interfaces (see Section 2.2).

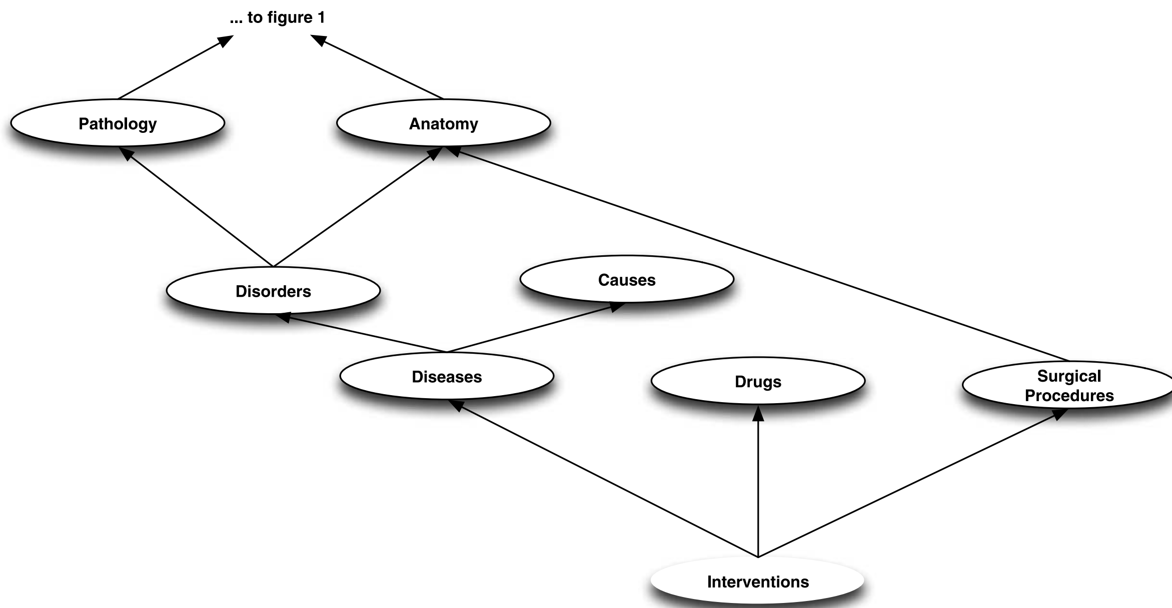


Fig. 2. Normalized ontology made up of cascading primitive and bridging modules.

This axiom would be expected to occur in the *Disorder* module, which imports both the *Anatomy* and *Pathology* modules. The Diseases module bridges the Disorders and *Causes* module to add further detail, e.g.,

BRCA1pos_carcinoma_of_breast *EquivalentTo:*
Carcinoma_of_breast *that* is_associated_with *some* BRCA1_gene

(BRCa1 is a gene associated with a high risk of breast cancer, defined, for this simplified example, in the Causes module.)

The *Interventions* module bridges the *Diseases*, *Drugs*, and *Surgical procedures* modules and contains axioms such as:

Tamoxifen *SubClassOf:* is_used_to_treat *some* Carcinoma_of_breast

(Tamoxifen is a drug commonly used to treat breast cancer.)

Note also that the Surgical Procedures module imports the Anatomy module separately so that there are two routes by which the Anatomy module is imported by the Interventions module. As stated earlier, such multiple import paths do not cause problems using the OWL imports mechanism.

Developing the normalized ontology as a set of modules has the advantage that it allows individual modules to be developed separately and for alternative modules to be substituted easily. For example, the interventions and pathology modules for studies of diseases of bones and joints or for infectious diseases might be different from those for cancer.

The value of normalization for ontology development has more than been confirmed by experience, both of ourselves and others. For example, it is implicit in the “principle of orthogonality” (Smith, 2007) used by the OBO Foundry family of biological ontologies. It has also been used to refactor important axes of the Gene Ontology (Fernandez-Breis et al., 2010; Wroe et al., 2003).

### 2.2. *Interfaces, binding and placeholders*

#### 2.2.1. *Interfacing ontologies*

In the previous section we stated: "*as long as none of the classes used in the bridging modules are removed or altered*, the individual modules can be updated separately". How are we to know which are the critical axioms that should not be altered, whether within a collaborative development team or when importing ontologies?

In programming, the notion of an Application Programming Interface (API) proved a breakthrough in collaborative software development and re-use. APIs allow developers to separate applications' public interfaces from their detailed internal structure and operation. They also help to focus developers' attention on providing clean sets of methods that others can understand in order to re-use their code.

Analogous notions have not yet been codified for ontology development, although our experience is that similar considerations apply to large ontologies and ontology driven software as to large software projects.

In ontologies, typically an importing module references what it regards as one or more top-level entities from an external module, and the subclasses of those entities are supplied by the imported module. These top-level entities may, or may not, already exist as named classes in the module to be imported and may, or may not, have the same identifiers in the importing modules. The referenced classes may be "top-level" only from the point of view of the importing module. They might be quite fine-grained notions in the imported module.

This gives rise to two issues:

- The importing ontology must indicate what top-level entities it needs to reference. We term the collection of these entities the "interface-submodule" of the importing module. The entities in it we refer to as "placeholders", because, typically, they have no definition in the importing module but will be defined by the imported modules. We usually find it convenient to implement the "interface-submodule" as a separate file although it can be indicated virtually by means of annotations or namespaces.
- The entities in the imported modules must be bound to the placeholders in the interface module by equivalence or subclass axioms. We term these axioms "binding axioms" and normally localise them to a "binding module". It is the "binding module" that adapts the imported modules to the needs of the importing module.

In the example shown in Fig. 3 and Table 2, the importing ontology – the disease module – requires access to the classes *Disease* and *Anatomic_structure.* However, the modules to be imported provide no single class for *Disease*, but rather two classes: *Lesion* and *Pathological_process* (both in the Pathology module). The placeholder *Disease* must be bound to their disjunction with an EquivalentTo: axiom as shown in Fig. 3. More simply, the importing ontology's class *Anatomic_structure* must be bound by an EquivalentTo: axiom to *Material_anatomic_entity* in the imported ontology.

Usually the binding axioms are formulated as EquivalentTo: axioms, but in some cases, SubClassOf: axioms are more appropriate. For example, the disjunction in Fig. 3 – Lesion or Pathological_process – could be weakened by replacing it with a pair of subclass axioms:

Lesion *SubClassOf:* Disease
Pathological_process *SubClassOf:* Disease.

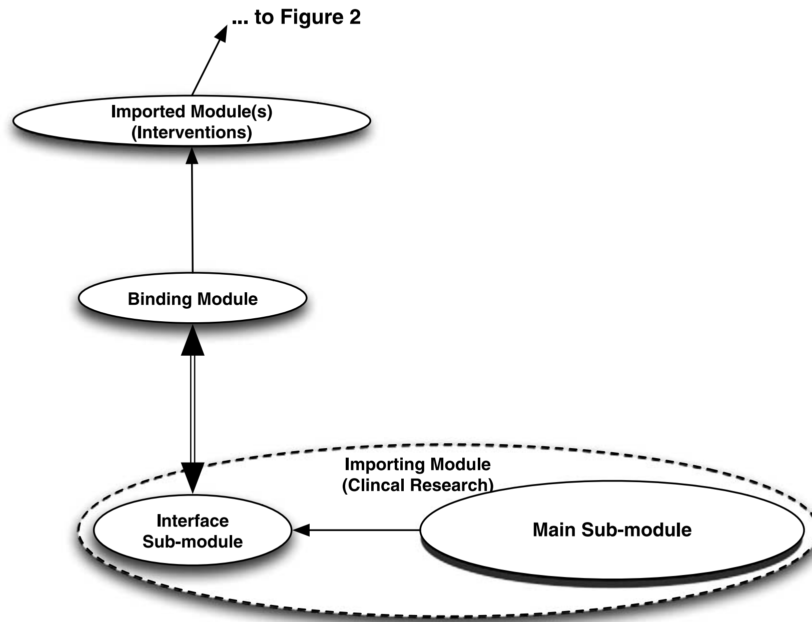


Fig. 3. Interface and binding modules for ontologies.

Table 2
Example contents of modules in Fig. 3

| Module | Example axioms |
|---|---|
| *Imported module* – and its import closure (interventions from Fig. 2) | *Class: Pathological_process*<br>*Class: Lesion*<br>*Class:* Material_anatomic_entity |
| Binding module | Disease *EquivalentTo*:<br>Pathological_process *or* Lesion<br>Anatomic_structure *EquivalentTo*:<br>Material_anatomic_entity |
| Interface module | *Class:* Disease<br>*Class:* Anatomic_structure |
| Main module | Axioms some of which use the classes Disease and Anatomic_structure |

Using multiple subclass axioms leaves open the possibility that there might be other kinds of *Diseases*, perhaps imported from other modules.

The double arrow between the interface sub-module and the binding module indicates a bi-directional import. The binding module imports the interface sub-module in order to be able to reference the place-holder entities; the interface sub-module imports the binding axioms, which are in turn imported by the main importing module, in this case the *Disease* module.

### 2.2.2. *Ontology driven software and ontology binding interface*

One of our goals is to produce Ontology Driven Software Architectures analogous to Model Driven Architectures. In Ontology Driven Architectures, major application data structures and behavior are

derived from the ontology. This may be achieved either by generating data structures and programs based on the ontology (static/early binding) or by using the ontology at run time to control the generation of data structures and behaviors (dynamic/late binding).

For example, the software might be aware of, and have special provision for, the fact that anatomic entities have parts, should be displayed in a partonomy hierarchy, and that the ontology contained only the "upwards" *is_part_of* links, rather than the downward *has_part* links (the real situation in most anatomy ontologies). Provided that the notion of *Anatomic_entity* and a set of relations for partonomy are defined, the application can define generic objects (e.g., Java classes) to display or gather information for *Anatomic_entities* with the appropriate behaviors, but be neutral as to the content of the ontology and can derive the necessary data structures and content from it. As a second example, different kinds of diseases have different descriptors – infections can be chronic or acute, cancers have stages and, furthermore, different cancers have different staging schemes. An application can have a generic disease object implementing the notion of descriptors and take the specifics from the ontology. (For details of an example of one variant, see Pulestin et al., 2008a.)

Once ontologies become an integral part of software, it becomes essential to provide a stable interface for both software and ontology developers between the ontology and software. Software engineers expect stable APIs between software modules. They likewise require stable interface between software and ontologies, which we term an "Ontology Programming Interface" or "OPI" by analogy with "API". Without a stable interface between the ontology and the software, there is an overly tight coupling between the ontology and software that restricts the development of both.

The methods from the previous section can be adapted to meet this requirement, as shown in Fig. 4. Typically, there are a small number of key high level classes and properties in the ontology that are referenced directly by the application. Encapsulating these to a separate module provides the necessary decoupling analogous to that provided by an API. The Ontology Programming Interface specifies the entities – classes, properties and individuals (and sometimes some axioms specifying their interactions) – that must therefore remain stable. In many cases, the software then implements an Applications Programming interface to encapsulate access to the Ontology Programming Interface. Once the Ontology Programming Interface module and API are in place, the Ontology Programming Interface module can be set to import a minimal ontology module, or "stub", for testing the software, just as stubs are used in other forms of software development. When the ontology is suitably developed, the stub can be replaced with the intended ontology modules.

As in Section 2.2.1, if externally generated ontologies are used, it may be necessary to provide an Ontology Binding Module to adapt the external module to the conventions of the Ontology Programming Interface. Such a module is included in Fig. 4 and Table 3.

### 2.3. Localisation

There are often general schemas, rules and policies at the enterprise level that must be specialised for use at the local level. For example, there might be a generic enterprise policy and generic rules for what to do in the case of "elevated blood pressure". However, different sites might have different criteria of what constituted an "elevated blood pressure".

Such local variations in both time and space are ubiquitous in the medical domain. For example, the threshold for diagnosis of Diabetes Type II using a glucose tolerance test in the UK (and most other communities) has been lowered several times in the past few years (see, e.g., Barr et al., 2002).

Similarly, the criteria for what constitutes "obesity" is highly controversial (see, e.g., Heiat et al., 2001). Furthermore, normal ranges for many biological tests vary between laboratories and even from

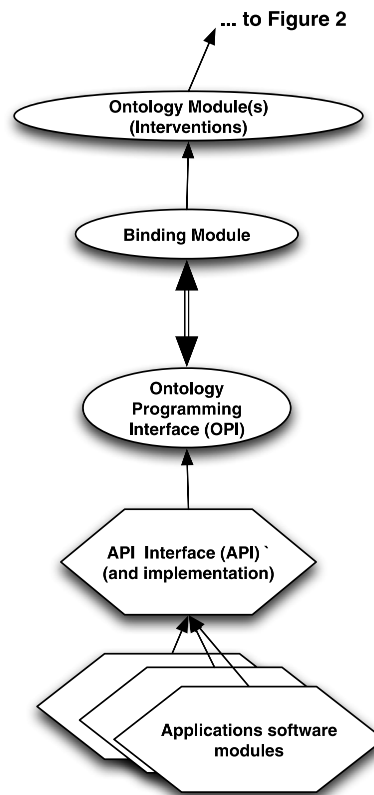


Fig. 4. Ontology Programming Interface for ontology driven software.

Table 3

Example contents of modules in Fig. 4

| Module | Example axioms |
|---|---|
| *Ontology modules* – and its import closure (interventions from Fig. 2) | *Class:* Pathological_process<br>*Class:* Lesion<br>*Property:* has_stage<br>*Property:* has_marker<br>*Class:* Material_anatomic_entity<br>*Property:* is_part_of |
| Binding module | Disease *EquivalentTo*: Pathological_process *or* Lesion<br>has_stage *SubPropertyOf:* has_descriptor<br>has_marker *SubPropertyOf:* has_descriptor<br>Anatomic_structure *EquivalentTo:* Material_anatomic_entity<br>*Property:* is_part_of transitive |
| Ontology Programming Interface (OPI) | *Class:* Disease<br>*Property:* has_descriptor<br>*Class:* Anatomic_structure<br>*Property:* is_part_of |
| API (& implementation) | Specification of Java classes and methods to access and use entities in OPI (and implementation) |

time to time in the same laboratory. Units of measure used in different institutions also vary. Although SI units (NIST, 2000) are considered official across much of the world, many institutions find older units convenient – e.g., many US institutions express weight and height in pounds, feet, and inches rather than kilograms and meters.

In most cases, once the interpretation of the raw data has been performed, the inferences and rules are relatively stable, e.g., the rules for what to do in response to an elevated blood pressure reading are likely to remain the same, regardless of what the threshold for "elevated" may be. Furthermore, the clinical management rules change asynchronously with the thresholds and measurements. Therefore, it is highly desirable to separate them.

This can be done using a variant of Binding Modules as described in Section 2.1. In this case the generic terms – e.g., elevated blood pressure – act as placeholders did in Section 2.1. Their detailed definition is given in a Localisation module as shown in Fig. 5 and Table 4. In this case, the local applications see the ontology through the "lens" of the localization module. (The details of rule languages and their interaction with OWL are outside the scope of this paper. It suffices for our purposes to consider them as simply another module.)

Other aspects of localization in medical applications include local synonyms, acronyms, abbreviations, etc., that need to be recognized and in some cases displayed differently in different locations or at different times. In general, this information is represented as annotations rather than logical axioms, but most of the same general principles apply to their modularisation. However, while OWL description logic semantics determine how logical axioms interact, it specifies no standard behavior when annotations interact – e.g., when there is more than one label for an entity. Applications programs or tools must implement some strategy to manage such conflict.

### 2.4. *Ontology extension and encapsulation of modifications*

It is often useful to be able to extend a module without disturbing the original but still to be able to visualize and classify the combined original and its extension. This can be achieved by interposing the new module at precisely the correct place in the imports graph. However, this can be cumbersome and error prone. It is often easier to add the new module as an "extension" of the old and then to have the two modules mutually import each other. This is particularly convenient when working collaboratively, and one collaborator wants to experiment with the work of another with a minimum of conflict.

An example is shown in Fig. 6 and Table 5. In Fig. 6, the anatomy module is based on anatomists' view that anatomy as strictly a matter of structure and derivation during embryonic development. If an author wishes to add a functional view without disturbing the existing structures, it is possible to define an extension within which to define new classes and the relations between them and their parts, in this example *Cardio-respiratory system*, *Respiratory system* and *Cardiovascular system*. In such cases we typically also define a new sub-property, in this case *is_part_of_system,* because if the extension is merged with the original, the distinction between the original *is_part_of relation* and the added *is_part_of_system* property may be important. However, strictly speaking, this is not necessary.

If an extension is later tested and agreed, it is often, but not necessarily, merged into the ontology that it extends.

Another important use for extensions is to implement test cases to show that the intended inferences are indeed made. In this case, the extension will contain what we term "probe classes" with annotations that serve a similar function to unit tests in software. For example, disjointness can be checked by defining a probe class that is the intersection of two classes intended to be disjoint with an annotation that it should be inferred to be unsatisfiable. Obviously, it is not desirable to include probe classes in the

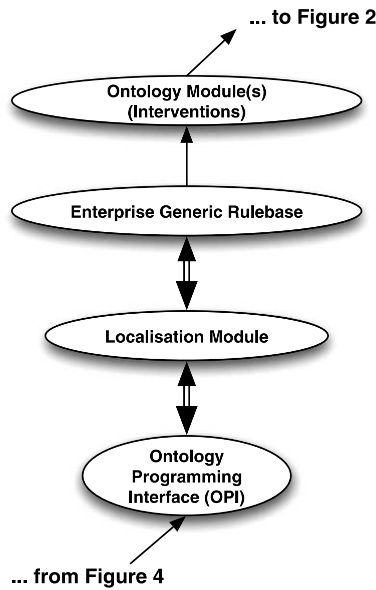


Fig. 5. Use of a localisation module.

Table 4
Example contents of modules in Fig. 5

| Module | Example axioms |
|---|---|
| Enterprise generic rulebase | *Class:* Elevated_blood_pressure<br>*Class:* Hypertension<br>*Rule:* Elevated_blood_pressure → Consider(Hypertension)<br>{. . .rules for considering hypertension. . .} |
| Localisation module | Elevated_blood_pressure *EquivalentTo*:<br>Blood_pressure *that*<br>((has_component *some* (Systolic that has_magnitude *some int[>=140]*))<br>*or*<br>(has_component *some* (Diastolic that has_magnitude *some int[>=90]*))) |

deployed ontology. If encapsulated in an extension module, the module can easily be omitted when the ontology is deployed.

## 3. Axiom-oriented vs. object-oriented mechanisms

As stated in the introduction, OWL imports are axiom-oriented rather than object-oriented. Although OWL ontologies are often presented as if they were a collection of objects – classes, properties and individuals – the object view is a user convenience imposed by tools over a set of unordered axioms. When one OWL module imports another, it simply aggregates the axioms from the imported module with those in the importing module.

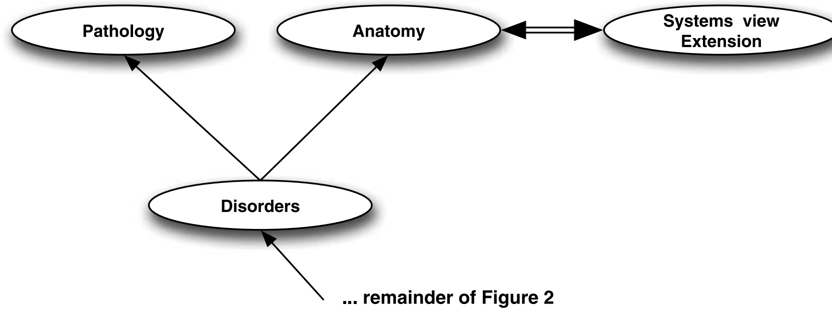


Fig. 6. Extensions to Anatomy module in Fig. 2 to add "systems view".

Table 5

Example contents of modules in Fig. 6

| Module | Example axioms |
|---|---|
| Anatomy | *Class:* Heart *SubClassOf:* Organ<br>*Class:* Lung *SubClassOf:* Organ<br>*Property:* is_part_of *transitive* |
| Systems view extension | *Class:* Body_system<br>*Property:* is_part_of_system *transitive*<br>*SubPropertyOf* is_part_of<br>*Class:* Cardiorespiratory_system<br>*SubClassOf:* Body_system<br>*Class:* Cardiovascular_system<br>*SubClassOf:* Body_system<br>*SubClassOf:* is_part_of_system *some* Cardiorespiratory_system<br>*Class:* Respiratory_system<br>*SubClassOf:* Body_system<br>*SubClassOf:* is_part_of_system *some* Cardiorespiratory_system<br>Heart *SubClassOf:*<br>is_part_of_system *some* Cardiovascular_system<br>Lung *SubClassOf:*<br>is_part_of_system *some* Respiratory_system |

### 3.1. Advantages of the axiom-oriented approach

There are three great advantages to the axiom-oriented approach. Firstly, it is easy for an importing module to add new information to a class from the imported module or vice versa. If there is information about the same entity in two modules, then any module that imports both modules will contain all the axioms about that entity, in effect merging the definitions and descriptions. The issues encountered object-oriented programming languages concerning altering the internal workings of an imported module do not apply.

Secondly, the order of imports is immaterial, since the result will simply be the aggregation of all the axioms in all the modules in the import tree. Therefore, cyclical imports as used in Figs 3, 5 and 6 pose no problems.

OWL's semantics naturally specify the interaction of logical axioms in the importing and imported ontologies. If they conflict, some class will be unsatisfiable or not classified as intended. In extreme

cases, the entire combined ontology may be unsatisfiable. No special attention is required authors or software engineers. (The same is not true of annotation axioms.)

### 3.2. *Disadvantages of the axiom-oriented approach*

#### 3.2.1. *What ontology does an entity "belong to"?*

Most users find it important to be able to think of each entity and axiom as "belonging" to a given module. From the users' point of view, most entities are introduced and belong to a specific module, even though they may also have additions from other modules. As in modular software development, refactoring the module structure is a common task. Furthermore, it is important to be able to locate the module in which an axiom appears in order to modify it.

However, strictly speaking there is no notion of an entity – class, individual or property – belonging to a module in OWL. Furthermore, OWL does not contain the notion of an axiom "occurrence" in a module; if the same axiom is asserted in two different modules, OWL treats it as just one axiom. Therefore, additional conventions are required if tools are to provide an "object oriented" or "frame oriented" view and to keep track of the source of entities and axioms in modules.

A first intuitive heuristic is that an entity belongs to the module in which it is "introduced" – e.g., the module containing the first subclass or equivalence class axiom encountered in the import tree. This notion is commonly used. However, unfortunately, since the import tree can be re-arranged, it is not reliable. A second possible criterion is to identify the module to which an entity belongs by its base IRI, but since any module may contain entities with different base IRIs, this too is unreliable. Furthermore, since IRIs serve as primary identifiers for OWL entities, if "moving" an entity between modules involves changing its IRI, then this will break any external references to the entity. Finally, OWL axioms do not have IRIs or any other form of identifier.

The only remaining option is to provide an annotation on each entity and axiom indicating the module to which it belongs. In OWL 1, this could be done for entities but not for axioms, but in OWL 2, both axioms and entities may be annotated.

However, whatever conventions are established by communities to identify entities and axioms with modules, they are not yet part of the standard, and remain a matter for each tool to implement its own solutions. For example, Protégé 4 allows axioms to be moved easily between modules – in effect to be cut from one file and pasted into another – but has no implementation of the notion of an entity, as opposed to axioms, belonging to a specific module. Until standards emerge, modular development is likely to work well only within communities using the same tools.

#### 3.2.2. *Merging ontologies*

Users likewise often want to merge modules, particularly if taking advantage of the "Extensions" mechanism to test modifications (Section 2.4). Excluding the issue of IRIs and any indication of conventions concerning what belongs to which module, merging of modules is straightforward – the merged module is simply the union of the axioms in the two modules to be merged. However, if there is to be a notion of entities belonging to modules or if IRIs are to be kept uniform within modules, then further work is required. Two cases should be distinguished: (a) pre-publication/pre-commitment experiments and development, where IRIs may be changed freely to harmonise the IRIs in the merged ontology; (b) post-publication, when it is necessary to preserve stable IRIs because there may be external references to them. To the best of our knowledge, no current tools separate the two cases cleanly.

#### *3.2.3. "Over-riding" or filtering out axioms: "Adapting" rather than just importing*

In most software systems, an importing module can over-ride a method from an imported module. In OWL, over-riding would correspond to the ability to filter out axioms. Since OWL is monotonic, this is not feasible directly within the paradigm. However, it is one of users' most frequent requests. For example, users want an importing ontology to "move" a class in the asserted subclass hierarchy. To do so requires filtering out the original subclass axiom from the imported module and adding a new subclass in the importing module. Alternatively, users may want an extension module to modify an existing restriction in a subclass axiom without modifying the original imported ontology. The cleanest solution is to filter out the original axiom from the imported module and add a replacement in the importing module. We shall call the process of importing with filtering "adapting" or "adaptation" to distinguish it from the existing OWL import mechanism.

At a micro level, since OWL 2 allows individual axioms to be annotated, it is not difficult to imagine a convention to attach an annotation that indicates that an axiom is to be ignored. At a macro level, the possibility of "over-riding" an axiom in an adapting module makes adaptation non-monotonic and, therefore, gives rise to the possibility of conflicts between modules. Any tool to support filtering out of axioms by an adapting module would require mechanisms to report or resolve such conflicts when they arise.

Despite this difficulty, allowing an importing module to filter axioms out of the imported module has the advantage that users find it easy to understand the notion that each extension module encapsulates a set of net changes. This is important both for purposes of quality assurance and determination of intellectual property. It also would also make it easier to implement new strategies for loosely coupled collaborative development in which each editor works on a different extension and any conflicts are resolved when the extensions are merged.

Perhaps most importantly, such adaptive modules would provide a means to encapsulate multiple sets of suggested changes, e.g., for consideration by an editorial board. Representing and managing such collections of suggestions is now a significant topic for at least one large distributed development project – the latest revision of the International Classification of Diseases (ICD-11) (Ustun et al., 2007).

No tool currently implements this feature. It fits awkwardly with the existing OWL API, because the internal data structures do not have a one-to-one correspondence to axioms. It is, therefore, likely to be deferred for the immediate future. However, we have experimented with manual approximations using annotations and scripts and are convinced of the value and utility. Perhaps a separate construct such as "adapts" rather than "imports" should be proposed to distinguish between cases in which filtering is possible and when it is not.

The alternative is a system of "diffs" and "patches" analogous to many software engineering tools, which fits better with the OWL API and, arguably, software engineering practices. We have used a locally developed diff/patch tool (Drummond, 2010), but it seems less transparent, at least to some domain experts who are key users, and is cumbersome when there are many sets of changes to consider. Furthermore, it requires the ability to download and edit a local copy of the complete ontology, which is not always possible.

## 4. Other OWL issues and suggested conventions

### *4.1. Multiple uses of IRIs within OWL*

OWL uses IRIs both to indicate physical location on the Web and as identifiers. This makes troublesome the issue of what IRIs to use within different modules that form a set, particularly during the

development phase. Clearer policies are needed – e.g., an extension to Berners Lee's paper on "cool IRIs" (Berners Lee, 1998) that addresses the special issues in OWL.

### *4.2. How to refer to each modules within sets of modules*

OWL 1 named ontologies by their "base IRI". OWL 2 names ontologies by their physical IRI. This gives rise to difficulties in distributing sets of modules if they are to be used off-line, particularly during development. Protégé 4.1 addresses this by assuming that sets of OWL ontologies will be developed within a single directory, and adding a catalog.xml file that specifies the redirection of the IRIs. If appropriate conventions are followed, the catalog.xml file can be generated automatically; if not, it can be generated the first time the ontology is loaded off line by selecting the physical file location for each file manually. However, neither this nor any analogous mechanism is part of the standard.

### *4.3. Which module to load*

Receiving a set of a dozen or more modules as a directory of OWL files that somehow import each other can be daunting. All ontologies should come with documentation, but there is no standard for where that documentation should be – In a separate file named "documentation" or something similar? In a README file? In Release notes? In the ontology annotations of one or more modules? Furthermore, written documentation is of little use to software. In addition, none of the documentation mechanisms of which we are aware addresses the issue of modular structure specifically, e.g., OWLDoc in Protégé (Horridge, 2011), OWLDoc in NEON (Munos-Garcia & Garcia-Delgado, 2010) or LODE (Peroni, 2011).

We use and recommend the convention that the file that is intended to be loaded be given a distinctive name – we use "START-HERE.owl". This module contains nothing except the overall ontology annotations and documentation for the set and the import statements for the next set of modules in the import graph. This is analogous to the HTML convention for loading "index.html" if present whenever a folder is referred to. This convention is easily recognised by software and humans alike, self-explanatory, and allows the inclusion of additional files in the same set – e.g., alternative versions of some modules, optional extensions, test ontologies, etc. An alternative machine-readable mechanism would be an extension of the "catalog" mechanism described in Section 4.2 used by Protégé.

## 5. Discussion and practical experience

We have outlined a series of use cases for developing ontologies as modules using interfaces and binding modules as part of an ontology engineering approach to improve re-use of ontologies.

In practice, the use of modules for upper ontologies is well established (Section 2.1). The pattern for normalization of ontologies (Section 2.2) is an extension of previous work and now standard within our group for several years. In fact, most ontologies in our group are now built as normalized, modularized ontologies and distributed as sets of files with "START-HERE.owl" as the root module as described in Section 4.3.

More complex relations between ontologies, plus the desire to allow the plug-and-play style exchange of modules, brought with it the need to define the notions of placeholders and binding interfaces between ontologies. This led to the notion of an Ontology Interface Sub-module (Section 2.2.1), which has also been used in the development of collaborative ontologies such as the Ontology for Clinical Research

(OCRe) (Carini et al., 2009). Using ontologies to drive software gave rise to the issue of maintaining a stable interface between the software and ontology so as to allow each to evolve independently (Section 2.2.2). In both cases, the interface concepts have sometimes been encapsulated in separate modules and sometimes simply marked by annotations. The choice is largely a matter of details of tooling and software, and the effect is the same. For clarity, we have chosen to present all interfaces in this paper as separate modules.

Ontology Programming Interfaces for Ontology Driven Architectures are in active use in ongoing commercial collaborations on clinical information systems and formed a key part of an earlier project in clinical systems (Pulestin et al., 2008a, 2008b).

The localization methodology (Section 2.3) is one of the key features giving added value in the practical industrial collaboration where the enterprise ontology must defer the precise criteria for certain common generic concepts to local modules.

Despite the limitations imposed by the inability to filter out axioms, extension modules (Section 2.4) have become a standard part of our collaboration methodology and played a key role in recent work analysing the sources of errors and their possible repairs in the very large description-logic-based terminology, SNOMED CT ( Rector & Iannone, 2011; Rector et al., 2011a, 2011b).

The OWL 2 standard (W3C OWL Working Group, 2009), OWL API version 3 (Horridge et al., 2007; OWL API Developers, 2011), and Protégé 4 provide tools that address many of the issues raised. However, extensions are needed to give finer grained control over handling IRIs. Despite the difficulties, many users believe they would find mechanisms for filtering out axioms in "adapting" modules useful.

We choose to implement all modules as separate files. As indicated in the introduction, alternative approaches are possible. For example, modules might be distinguished by annotations, IRI conventions, namespaces or similar mechanisms. However, if we wish to replace one version of a module with another or share common modules between multiple ontologies, none of these mechanisms are as convenient as using files and imports. All other mechanisms depend on special extensions in tools whereas the mechanism of using individual files depends primarily on the import mechanism defined in the OWL standard.

It is, of course, not necessary to import an entire ontology in order to refer to an entity within it – a simple reference to the entity's IRI is all that is required. However, such mention without importing does not give the referencing ontology any information on the semantics associated with the referenced entity that may have been implemented in the originating ontology. If there are axioms in the source ontology that are intended to be interpreted by a reasoner, the results will almost certainly not be what the author of the source ontology intended or other users of the source ontology experience.

Recently, we have been experimenting with combining the methods for extracting modules based on signatures with the methods for constructing and extending ontologies using modules as described in this paper. These mechanisms are particularly helpful when dealing with very large ontologies such as the medical terminology SNOMED-CT (Stearns et al., 2001), which contains over 300,000 classes. Identifying the signature for a problematic area, extracting a module, then further extracting the submodules for anatomy, and then normalizing them provides a means of rationalizing complex subfields such as head injury that have presented difficulties. Similarly, performing experiments in encapsulated extensions has proved helpful in communicating the suggested changes amongst authors and editors (Rector & Iannone, 2011; Rector et al., 2011a, 2011b).

The examples in this paper are all taken from biomedicine. It remains for others to test to what extent they methods generalize, or whether ontologies for other domains will raise similar issues. However, nothing in the structure of the methods or use cases presented here is specific to biomedicine.

Throughout, our primary concern is to facilitate an engineering methodology for developing large ontologies covering many different subdomains and making significant use of inference. On the basis of our experience to date, modular development is likely to be an important tool for this purpose.

## Acknowledgements

This work was supported in part by the JISC and UK EPSRC projects CO-ODE and HyOntUse (GR/S44686/1), the EU funded Semantic Mining Network of Excellence and EU funded SemanticHEALTH FP6 Specific Support Action, IST-27328-SSA, http://www.semantichealth.org, and Siemens Healthcare Systems. The collaboration of the members of the Ontologies for Clinical Research (OCRe) consortium and the WHO ICD-11 Health Information Modelling Topic Advisory Group (HIMTAG) is gratefully acknowledged.

## References

Aneurist Consortium, http://www.aneurist.org/ (accessed 3 March 2011).

Bao, J. Caragea, D. & Honavar, V. (2006). Towards collaborative environments for ontology construction and sharing. In *Proceedings of the International Symposium on Collaborative Technologies and Systems* (pp. 99–108). Los Alamitos, CA, USA: IEEE Computer Society.

Barr, R.G., Nathan, D.M., Meigs, J.B. & Singer, D.E. (2002). Tests of glycemia for the diagnosis of type 2 diabetes mellitus. *Annals of Internal Medicine*, *137*, 263.

Berners Lee, T. (1998). Cool URIs don't change. Available at: http://www.w3.org/Provider/Style/URI.html (accessed December 2010).

Bootstrep Consortium. Bootstrep homepage, http://www.bootstrep.org/bin/view/Extern/WebHome (accessed 3 March 2011).

Carini, S., Pollock, B.H., Lehmann, H.P. et al. (2009). Development and evaluation of a study design typology for human research. In *Proceedings of the AMIA Annual Symposium on American Medical Informatics Association*, San Francisco, CA (pp. 81–85). Bethesda, MD: AMIA.

Cimino, J.J. (1998). Desiderata for controlled medical vocabularies in the twenty-first century. *Methods of Information in Medicine*, *37*, 394–403.

College of American Pathologists. SNOMED homepage, www.snomed.org (accessed 2 October 2010).

de Coronado, S., Haber, M.W., Sioutos, N., Tuttle, M.S. & Wright, L. (2004). NCI Thesaurus: using science-based terminology to integrate cancer research results. In *Proceedings of the World Conference on Medical Informatics*, San Francisco, CA (pp. 33–37). Amsterdam: IOS Press.

Debugit Consortium. Debugit homepage, http://www.debugit.eu/ (accessed 3 March 2011).

Del Vescovo, C., Parsia, B., Sattler, U. & Schneider, T. (2010). The modular structure of an ontology: an empirical study. In *Proceedings of the Conference on Modular Ontologies: Proceedings of the 4th International Workshop* (pp. 11–24). Amsterdam: IOS Press.

Drummond, N. (2010). Manchester Description Logic Group. OWL Patch. Available at: http://owl.cs.manchester.ac.uk/patch/ (accessed October 2010).

Ensan, F. & Du, W. (2008). An interface-based ontology modularization framework for knowledge encapsulation. In *Proceedings of the Semantic Web-ISWC* (pp. 517–532). Berlin: Springer.

Fernandez-Breis, J.T., Iannone, L., Palmisano, I., Rector, A. & Stevens, R. (2010). Enriching the Gene Ontology via the dissection of labels using the Ontology Pre-Processor Language. In *Proceedings of the European Knowledge Acquisition Workshop*. Lecture Notes in Computer Science (Vol. 6317, pp. 59–73). Berlin: Springer.

Grau, B.C., Horrocks, I., Kazakov, Y. & Sattler, U. (2008). Modular reuse of ontologies: theory and practice. *Journal of Artificial Intelligence Research*, *31*, 273–318.

Hasse, P., Lewen, L. & Studer, R. (2008). The NeOn ontology engineering toolkit. In *Proceedings of the WWW2008*, Bejing, China. Available at: citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.4163&rep=rep1&type=pdf.

Heiat, A., Vaccarino, V. & Krumholz, H.M. (2001). An evidence-based assessment of federal guidelines for overweight and obesity as they apply to elderly persons. *Archives of Internal Medicine*, *161*, 1194.

International Health Terminology Standards Development Organisation (IHTSDO). Homepage, http://www.ihtsdo.org/ (accessed 13 February 2011).

Horridge, M. Bechhofer, S. & Noppens, O. (2007). Igniting the OWL 1.1 touch paper: the OWL API. In *Proceedings of the OWL Experiences and Directions*. CEUR (Vol. 258). Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.4920&rep=rep1&type=pdf.

Horridge, M. (2011). OWLDoc (CO-ODE plugin for protege). Available at: http://code.google.com/p/co-ode-owl-plugins/wiki/OWLDoc (accessed 3 October 2011).

Horridge, M., Drummond, N., Knublauch, H. et al. (2004). Building ontologies with protege-OWL plugin. Available at: http://www.co-ode.org/resources/.

Isaac, A. & Summers, E. (2009). SKOS simple knowledge organization system primer. Available at: http://www.w3.org/TR/skos-primer/.

Munos-Garcia, O. & Garcia-Delgado, A. (2010). OWLDoc (in NEON toolkit). Available at: http://neon-toolkit.org/wiki/OWLDoc (accessed 28 September 2011).

NeOn Consortium (2011). NeOn Toolkit. Available at: http://semanticweb.org/wiki/NeOn_Toolkit (accessed 4 May 2011).

NIST (2000). International System of Units (SI) home page. Available at: http://physics.nist.gov/cuu/Units/ (accessed 10 October 2011).

OWL 2 Web Ontology Language Profiles (2009), http://www.w3.org/TR/owl2-profiles/ (accessed 5 March 2011).

OWL API Developers (2011). OWL API homepage, http://owlapi.sourceforge.net/ (accessed 2 February 2011).

Patel-Schneider, P.F., Hayes, P. & Horrocks, I. (2004). OWL Web Ontology Language: semantics and abstract syntax. Available at: http://www.w3.org/TR/2004/REC-owl-semantics-20040210/.

Pathak, J., Johnson, T.M. & Chute, C.G. (2009). Survey of modular ontology techniques and their applications in the biomedical domain. *Integrated Computer-Aided Engineering*, *16*, 225–242.

Peroni, S. (2011). Live OWL documentation environment (LODE). Available at: http://www.essepuntato.it/lode, http://sourceforge.net/progjects/lode (accessed 5 Oct 2011).

Pulestin, C., Cunningham, H. & Rector, A. (2008b). A generic software framework for building hybrid ontology-backed models for driving applications. In *Proceedings of the OWL Experiences and Directions*, Washington, DC.

Pulestin, C., Parsia, B., Cunningham, J. & Rector, A. (2008a). Building hybrid ontology-backed software models. In *Proceedings of the International Conference on Semantic Web*, Karlsruhe, Germany. Lecture Notes in Computer Science (Vol. 5318, pp. 130–145). Berlin: Springer.

Rector, A. (2003). Modularisation of domain ontologies implemented in description logics and related formalisms including OWL. In *Proceedings of the Knowledge Capture*, Sanibel Island, FL (pp. 121–128). New York: ACM Press.

Rector, A., Brandt, S. & Schneider, T. (2011a). Getting the foot out of the pelvis: modelling problems affecting use of SNOMED CT hierarchies in practical applications. *JAMIA*, *18*, 432–440.

Rector, A. & Iannone, L. (2012). Lexically suggest, logically define: quality assurance of the Use of qualifiers and expected results of post-coordination in SNOMED CT. *Journal of Biomedical Informatics*, *45*(2), 199–209.

Rector, A., Iannone, L. & Stevens, R. (2011b). Quality assurance of the content of a large DL-based terminology using mixed lexical and semantic criteria: experience with SNOMED CT. In *Proceedings of the K-CAP*, Banff, Canada (pp. 57–64). New York: ACM Press.

Rector, A., Horridge, M., Iannone, L. & Drummond, N. (2008). Use cases for building OWL ontologies as modules: localizing, ontology and programming interfaces and extensions. In *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering*, Karlsruhe, Germany (Best Paper Award). Available at: http://www.abdn.ac.uk/~r01srt7/swese2008/pdf/swese2008_submission_6.pdf.

Rosse, C., Shapiro, I.G. & Brinkley, J.F. (1998). The digital anatomist foundational model: principles for defining and structuring its concept domain. *Journal of the American Medical Informatics Association* (AMIA'98 Symp. Suppl.), 820–824.

Sattler, U., Schneider, T. & Zakharyaschev, M. (2009). Which kind of module should I extract? In *Proceedings of the of DL-2009*. CEUR (Vol. 477). Available at: CEUR-WS.org/Vol-477/.

Schulz, S., Beisswanger, E., Hahn, U., Wermter, J., Stenzhorn, H. & Kumar, A. (2006). From GENIA to BioTop – towards a top-level ontology for biology. In *Proceedings of the of the International Conference on Formal Ontology in Information Systems*, Baltimore, MD, USA (pp. 103–114). Amsterdam: IOS Press.

Seidenberg, J. & Rector, A. (2007). The state of multi-user ontology engineering. In *Proceedings of the 2nd International Workshop on Modular Ontologies at KCAP'2007*, Whistler, Canada. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.8802&rep=rep1&type=pdf.

Sim, I., Carini, S., Tu, S. et al. (2010). The human studies database project: federating human studies design data using the Ontology of Clinical Research. In *Proceedings of the 2011 AMIA Summit on Clinical Research Informatics*, San Francisco, CA (pp. 51–55). Bethesda, MD: AMIA.

Smith, B. (2007). The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration. *Nature Biotechnology*, *25*, 1251–1255.

Stearns, M.Q., Price, C., Spackman, K.A. & Wang, A.Y. (2001). SNOMED clinical terms: overview of the development process and project status. In *Proceedings of the AMIA Fall Symposium* (pp. 662–666). Philadelphia, PA: Henley & Belfus.

Thomas, C.J., Sheth, A.P. & York, W.S. (2006). Modular ontology design using canonical building blocks in the biochemistry domain. In *Proceedings of the 4th International Conference on Formal Ontologies in Information Systems* (pp. 115–127). Amsterdam: IOS Press.

TopQuadrant (2011). TopBraid Composer home page. Available at: http://www.topquadrant.com/products/TB_Composer.html (accessed 3 October 2011).

Ustun, T.B., Jakaob, R., Celik, C. et al. (2007). Production of ICD-11: the overall revision process. Available at: www.who.int/classifications/icd/ICDRevision.pdf (accessed 3 March 2011).

W3C OWL Working Group (2009). OWL 2 Web Ontology Language: document overview. Available at: http://www.w3.org/2007/OWL/wiki/Document_Overview (accessed 2 October 2010).

Weibel, S., Kunze, J., Lagoze, C. & Wolf, M. (1998). Dublin core metadata for resource discovery. In *Internet Engineering Task Force RFC* (Vol. 2413). San Francisco, CA: University of California.

Wroe, C.J., Stevens, R., Goble, C.A. & Ashburner, M. (2003). An evolutionary methodology to migrate the Gene Ontology to a description logic environment using DAML+OIL. In *Proceedings of the 8th Pacific Symposium on Biocomputing*, Hawaii (pp. 624–635). Stanford, CA: Stanford University.