Building Modular Ontologies and Specifying Ontology Joining, Binding, Localizing and Programming Interfaces in Ontologies Implemented in OWL

Alan Rector, Matthew Horridge, Nick Drummond

School of Computer Science, University of Manchester,
Manchester M13 9PL, UK
rector@cs.manchester.ac.uk

Abstract

The notion of an Application Programming Interface (API) proved a breakthrough in software modularization and re-use by allowing developers to separate applications' public interfaces from their detailed internal structure. No comparable notion exists currently for ontologies, although there is considerable other work on issues related to modularization of ontologies. We present four use cases for "Ontology Programming Interfaces" and discuss the existing features of OWL that facilitate this approach and the additional features needed to consolidate it.

Introduction: The Notion of an "Ontology Programming Interface"

The notion of an Application Programming Interface (API) proved a breakthrough in software modularization 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 operations and methods to allow others to understand and re-use their code.

Currently, no comparable notion exists for ontologies, even those designed to be used in ontology driven architectures. Existing studies of modularization have concentrated either on "conservative extensions" (Lutz et al., 2007), on structural criteria for modularization (Schlicht and Stuckenschmidt, 2006), on attempts to understand the import of different parts of the ontology (Pan et al., 2006), or on generic locking mechanisms (Seidenberg and Rector, 2005). In this paper, we take an somewhat different approach inspired by classic software engineering paradigms, in which modularity, interfaces between modules, and re-use are part of the basic ontology development process.

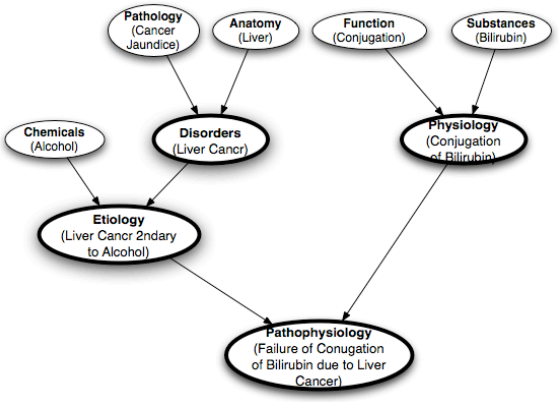
One reason for the shortage of studies on building modular ontologies is that most ontology development tools have made modularization difficult. In part, this is because the OWL specification on importing, naming, and versioning of ontologies is at best confusing and at worst incomplete. The work reported here has been made possible in large part because the new Protege4 OWL development environment[1] and the new OWL API (Horridge et al., 2007) have adopted conventions to make it much easier to develop OWL ontologies as sets of modules. Experience made possible by these new tools has led to the rapid development of a the notions of "binding", "joining", "interface", and "localizing" modules presented here.

This experience has also shown that OWL's structure as a set of global axioms rather than a set of encapsulated objects makes modularization extremely powerful, because it allows new axioms concerning existing entities to be added at any time to any module. However, in its raw form, the OWL paradigm runs counter to these users' expectations. Most users assume a more object-oriented point of view in which each entity is encapsulated in a single set of expressions that belong to a single module. Protégé-OWL, therefore, overlays the underlying OWL representation with an object-oriented view that discriminates between the various ways in which an entity can be referred to.

In the remainder of this paper, we first sketch the different use-cases for modularization and the interfaces associated with each. We then touch briefly on the conventions used by Protege4-OWL to facilitate development and use of modular ontologies. Note that all examples are given using the Manchester OWL syntax (Horridge et al., 2006)

[1] See . http://protege.stanford.edu/download

Figure 1: Normalisation & Joining Modules



Use Cases

Use Case 1: Ontology Normalization and Joining

In a previous paper, (Rector, 2003) we introduced the notion of a normalized ontology formed by a set of strict mono-hierarchical trees of primitive entities joined by definitions and descriptions. In that paper, both the primitive trees and the joining axioms were implemented in a single module, as at that time developing ontologies in separate modules was cumbersome at best.

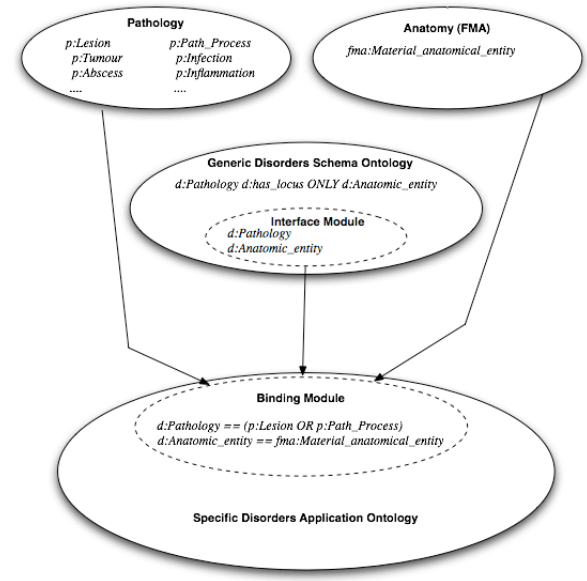
Subsequent experience has more than confirmed the value of normalization as an approach. Efficient means for modularization means that we can now implement each major tree – e.g. structure, function, disorder, causative agents, etc – in a separate module and then provide one or more "joining ontologies" that contain the axioms and definitions link them together. Figure 1 shows a cascade of such normalized and joining ontologies, with the joining ontologies shown in bold. The cascade allows the composition of complex notion out of careful factored individual ontologies, in this example that of a "*Cancer (disorder) of Liver (structure) secondary to Alcohol (causative agents) that impairs conjugation (function) of Bilirubin (structure) causing Jaundice (disorder)*".

Use case 2: Ontology Binding Interface & Placeholders

In many situations, a single core schema ontology is to be used with several alternative domain ontologies, *e.g.* a single ontology of the structure of clinical trials might be bound to a number of different disease and treatment ontologies, depending on the topic – *e.g.* cancer, infectious disease, congenital malformations, etc. In this case, the interface sub-ontology is the direct analogue of the API.

Consider the example in Figure 1. A single schema ontology of disorders might be used with several different ontologies of anatomy – one for surgical anatomy and an alternative one for developmental anatomy – and several alternative ontologies for pathology.

Figure 2: Schema and Ontology Binding Interface



What is required in this case is for the schema ontology to be able to define the domains and ranges of relations at a generic level, and then allow these to be bound to the specific notions for each subtopic.

Therefore, the it is necessary for the schema ontology to identify those key entities needed for its schema by "placeholders" classes. Equivalence axioms can bind the placeholder classes to the specific classes in an application ontology as required.

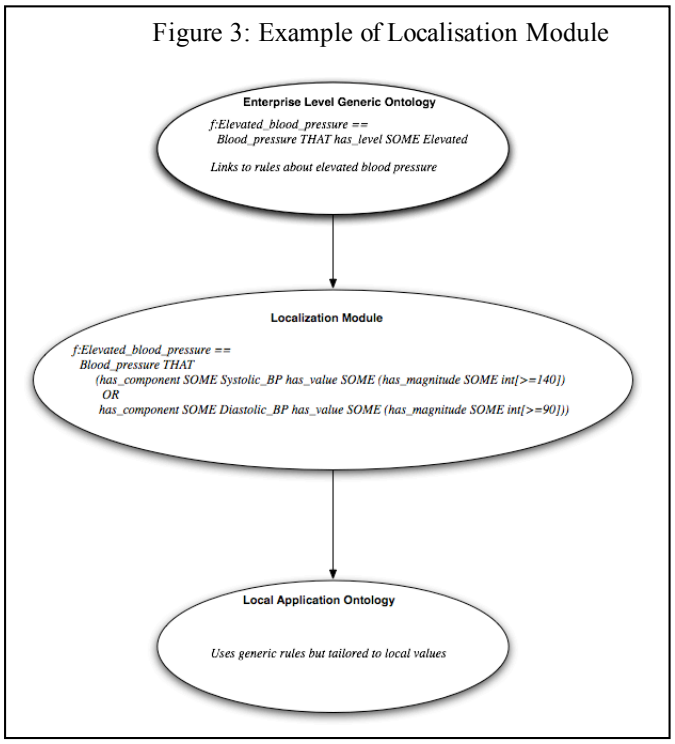
Figure 2 demonstrates the principle by showing the relations to the disorder module in Figure 1 in more detail. Prefixes are used for the namespaces for each ontology. The generic disorders schema ontology includes axioms defining the domain and range of properties that will link the imported Pathology and Anatomy pathology.[1] However, the disorder schema ontology makes minimal commitment to the nature of the anatomy or pathology ontologies or their contents.

To use the generic disorder schema, an application ontology must implement a binding ontology which defines the placeholders from the disorder schema – *d:Pathology* and *d:Anatomic_entity* in terms of the imported ontologies for anatomy and pathology. Usually the binding definitions are formulated as equivalence axioms, but in some cases, subclass axioms are more appropriate. For example, the disjunction in Figure 2 could be weakened by replacing it with a pair of subclass axioms, thus not foreclosing the possibility that other classes, perhaps from other ontologies, might be kinds of pathology.

[1] The "Foundational Model of Anatomy" (FMA)(Rosse et al., 1998) is assumed as an for anatomy.

In future implementations, it is hoped to implement a system of "packages" so that the Interface Module and Binding Module can be encapsulated within the schema and applications ontologies as implied by the dotted ovals in Figure 2. However, the current OWL standard does not support such formal nesting of ontologies or the control on visible and private entities that would naturally follow. In current implementations, therefore, all modules are at the same level.

Figure 3: Example of Localisation Module



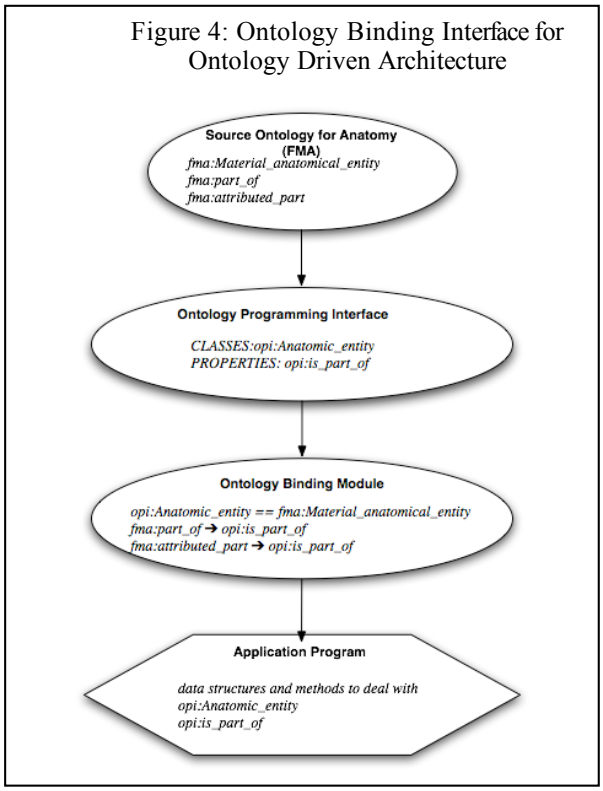
Use Case 3: Localization

In many situations, there are general schemas and policies at the organisation or enterprise level that must be specialized at the local level. For example, there might be a general enterprise wide policy and generic rules for what to do in cases of "elevated blood pressure." However, different departments' or sites might have different criteria for when a blood pressure is to be considered elevated. Variations between sites in normal ranges and thresholds are common, particularly new biological assays and genetic tests, but occur even with relatively common measurements. Furthermore, these thresholds change with time. For example, in the UK the policy for treating newly diagnosed type two diabetes has remained unchanged for several years, but the threshold for diagnosis has been repeatedly lowered throughout this period. Managing such local variations is a major task for many clinical systems.

Figure 3 sketches one solution to this problem. The generic axioms concerning elevated blood pressure reside in the *core* or *enterprise* ontology and associated with enterprise wide rules. The more specific axioms, *e.g.* the actual numeric thresholds, reside in a *localizing* ontology that is imported by applications along with the enterprise ontology.

Note that this strategy is a supplement rather than a replacement for more complex forms of abstraction. For example, a system might contain a rule that "A diagnosis of essential hypertension should be made only after three independent readings of an elevated blood pressure at rest and ruling out all specific causes including surgical and endocrine hypertension.)". The localizing ontology would allow this rule to be formulated in a general way without mentioning specific thresholds for "elevated blood pressure", but the rule itself is clearly beyond the scope of a simple ontology

Figure 4: Ontology Binding Interface for Ontology Driven Architecture



Use case 4: Ontology Programming Interface

In Ontology Driven Architectures, analogous to Model Driven Architectures, application data structures and behavior is derived ontology, either statically or dynamically. This produces a tight coupling between the application and the ontology that can restrict both their development. Typically, there are a small number of key high level classes and properties in the ontology that are referenced directly by the application. Confining these to a separate *application interface* sub-ontology, which is agreed to remain stable, provide the necessary decoupling analogous to that provided by an API.

This pattern is very close to use case 2, except that in this case the "Interface Module" defines those classes that

must be understood by the software as in some sense "special" rather than those that must be bound in an interface ontology. An example is shown in Figure 4.

Typically, there will be special behavior or data structures associated with the classes in the Ontology Programming Interface that can be specialized according to the information on their subclasses in the ontology. For example, the software might be aware of, and have special provision for, the fact that anatomic entities have parts and perhaps even for the transitivity of parthood. However, the specific part-whole relations required for different classes of body parts would reside in the ontology. Provided that the notion of *Anatomic_entity* and a set of relations for partonomy were defined, the application would be neutral as to the content of the ontology and could derive the necessary data structures and content from it. Note that in this architecture, it is necessary to have an Ontology Binding Interface module between the source ontology (the FMA in this case) and the application Ontology Programming Interface in order to organize the imports correctly.

Critical Features of OWL for Modularization & Binding

OWL Imports: Axioms and "objects"

Although OWL is often presented as if it were a collection of objects – classes, individuals, and properties – an OWL ontology actually consists simply of a collection of axioms about entities that do not have to be named before they can be referenced. Unlike a Java class, or a class in a typical frame system, there is no formal sense in which an OWL class "belongs" to a particular module. Any module can contain axioms about any class.

OWL's axiom oriented view has both advantages and disadvantages. The disadvantage is that, practically for organisational and housekeeping purposes, it is helpful, perhaps even necessary to identify a given class or property with the module in which it "originates."

On the other hand, OWL's view of an ontology as simply a flat collection of axioms makes it easy to add additional information to "existing" classes in new modules. It is even possible to have classes in one module referenced in another without important that module, simply by using the correct full name, including the URI, although, at least with current tools, such import structures are difficult to manage.

This feature is critical to the strategies described here. In each use case, a "placeholder" class is defined in the interface module to be used by the ontology schema or application ontology. A second "binding ontology" then imports both the interface module and some major domain ontology and provides a set of subclass or equivalence axioms to link the placeholders to their concrete representation in the domain ontology.

Conventions for an object-oriented View of OWL Modules

The advantage of OWL's axiomatic structure is that it is possible to create very flexible import strategies. The disadvantage is that it requires conventions and some extensions to make it effective.

Since OWL axioms themselves have no identifiers or URIs, there is no intrinsic way to tell where an axiom has come from once it has been loaded from an imported ontology. Without this information, sensible editing policies are almost impossible. This problem is being addressed in OWL 1.1[1] by allowing comments to appear on individual axioms. Conventions can then be established to use these axioms to indicate the module in which the axiom appears.

The larger problem is that to use such modular ontologies, it must be easy to create, load, build, and move between multiple ontologies in a single "project" or "set". Protege4[2] has adopted a number of additional conventions that include a strong notion of the "originating" ontology for an entity and for the set of axioms that constitute its "original description." It overloads the notion of "base URI" to act as an identifier for the ontology independent of its physical location. Extensions under development include a clear distinction between the module in which a class originates and other modules in which it is referenced, along with better tools for refactoring ontologies that allow entire "original descriptions" to be managed as units.

Cognitive load on ontology authors and the need for improved tooling.

Regardless of the tooling, maintaining information in multiple ontologies imposes a significant additional load on the ontology developers. The tools must make refactoring and modification of the module structure easy, so that the long-term advantages of the modular structure are not outweighed by the short term costs of developing in a modular fashion. Although the Protege4 tools provide some of the needed functions, refactoring and reorganization of modules is still too time consuming and difficult. The lack of adequate tooling remains a significant disincentive to the modularization, even to those who recognize and value its advantages.

Discussion

Re-use of ontologies is still in its infancy. Most work on ontology modularization has concentrated on breaking down of existing large ontologies into disjoint pieces notions such as conservative extensions. Other work as focused on strategies for multi-user development using locking.

---

[1] http://www.webont.org/owl/1.1/

[2] See . http://protege.stanford.edu/download

This paper takes a different perspective, looking at strategies for providing well defined interfaces between ontologies, analogous to APIs for programming languages, so that development can take place by different groups independently on either side of the interface, or even that one ontology might be substituted for another provided each could be bound to the same interface.

This approach seems particularly important in very large domains where no single also makes it possible to define generic schemas for linking major categories of information that are relatively neural as to the internal structure of those categories.

Experiments have included development of ontologies within several biological domains, although none has reached the level of widespread dissemination and testing.

Acknowledgements

This work 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. The collaboration of the members of the Ontologies for Clinical Research (OCRe) consortium is gratefully acknowledged.

References

Horridge, Matthew, S Bechhofer, and Olaf Noppens. 2007. Igniting the OWL 1.1 touch paper: The OWL API. OWL Experiences and Directions (OWLEd 2007)

Horridge, Matthew, Nick Drummond, J Goodwin, Alan Rector, R Stevens, and Hai Wang. 2006. The Manchester OWL syntax. OWL: Experiences and Directions (OWLED 06) http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS//Vol-216/submission_9.pdf.

Lutz, C, Dirk Walther, and F Wolter. 2007. Conservative extensions of expressive description logics. International Joint Conference on Artificial intelligence (IJCAI 07) 453-458.

Pan, Jeff Z, L Serafini, and Yuting Zhao. 2006. Semantic Import: An Approach for Partial Ontology Reuse. Workshop on Modular Ontologies (WoMO'06)

Rector, Alan. 2003. Modularisation of domain ontologies Implemented in description logics and related formalisms including OWL. Knowledge Capture 2003 121-128.

Rosse, C, I G Shapiro, and J F Brinkley. 1998. The Digital Anatomist foundational model: Principles for defining and structuring its concept domain. Journal of the American Medical Informatics Association no. 1998 Fall Symposium Special issue: 820-824.

Schlicht, A, and Stuckenschmidt, H. 2006. Towards stuctural criteria for ontology modularization. 1st International Workshop on Modular Ontologies, WoMO'06

Seidenberg, Julian, and A Rector. 2005. Techniques for segmenting large description logic ontologies. K-CAP Ontology management Workshop