

Designing User interfaces to Minimise Common Errors in Ontology Development: The CO-ODE and HyOntUse Projects

Alan L Rector¹, Nick Drummond¹, Matthew Horridge¹, Jeremy Rogers¹, Holger Knublauch²,
Robert Stevens¹, Hai Wang¹ and Chris Wroe¹

¹Department of Computer Science, University of Manchester, M13 9PL UK

²Section on Medical Informatics, Stanford University, Stanford, CA USA

Abstract

Understanding and using OWL can be difficult. Until recently, the difficulty was made, if anything, greater by user interfaces which were designed more to demonstrate OWL than to fit users requirements. The new interfaces being developed in the Protégé-OWL/CO-ODE/HyOntUse projects are based, to a large extent, on experience in tutorials, workshops, and practical development with users. They are designed to avoid the most common errors and pitfalls and make debugging easier and quicker. At the same time, they have proved to streamline many operations so as to make development much faster and easier for experienced ontology engineers.

1. Introduction

Most people find it difficult to understand the logical meaning and potential inferences in description logics (DLs), including OWL-DL. Few example ontologies on the web make extensive use of inference. Yet those, including the authors, that advocate the use of DLs generally, and OWL-DL in particular, believe that the use of inference to support ontologies is a major advance. In particular, the use of DLs brings major benefits in the quality of the ontologies developed, in their reusability, and in the effort required to maintain them. However, if these benefits are to be more widely realised, the practical understanding must be passed on to others.

While there are several initial guides to ontologies available, *e.g.* [1, 2] and numerous works on ontological principles, *e.g.* [1, 3, 4], there is little that examines the problems of learning to build logic based ontologies. Nor do they highlight the use of inference as supported by reasoners (or classifiers), which is a specific design goal of OWL-DL. There are tools for OWL and its predecessors *e.g.* OilEd¹[5], but until recently, few tools were user oriented, and most made common patterns tedious.

Over the past five years we have presented a series of tutorials, workshops and post-graduate modules, teaching people to use OWL-DL and its predecessors effectively. A central task has been to teach the participants how to understand and use DLs and reasoners effectively. This has

led to a list of common errors which have in turn played a major role in the design of new tools² being developed by the Protégé-OWL/CO-ODE/HyOntUse collaboration [6]. The errors themselves are discussed in more detail in a separate paper [7]. This paper discusses the approach to the user interfaces, which are still evolving rapidly. One important goal is to invite participation by a wider community in the discussion and requirements forum³.

Our guiding principles are to:

- Make the easy thing the right thing – to make the defaults the usual case
- Make it easy to do the right thing – to provide shortcuts for tedious tasks
- Make it easy to do everything – to provide mechanisms to help guide users through complex tasks
- Make it easy to see what has been done – to present everything together in one place and improve visualisation
- Make it easy to find what is wrong – to provide a series of debugging aids

Following these principles we are producing an interface which both avoids common errors and, as a side benefit, is much faster and easier even for experienced users. The total time to create certain ontologies has been cut significantly.

¹ <http://oiled.man.ac.uk>

² <http://protégé.Stanford.edu> → plugins → backends → OWL; <http://www.co-ode.org>

³ <http://www.co-ode.org/forum>

2. The Problems

OWL differs in important ways from systems that most users have encountered previously.

- It does not assume that differently named objects are different;
- It uses open world rather than closed world reasoning;
- Its domain and range constraints are axioms to be reasoned with rather than constraints to be checked;
- It contains both existential and universally qualified links ('restrictions');
- It distinguishes between defined and primitive classes – *i.e.* between sets of necessary and sufficient conditions and lists of individually necessary conditions.

These characteristics combine to make certain errors common:

1. Failure to make all information explicit – much that other systems assumed must be explicitly stated in OWL.
2. Mistaken use of universal rather than existential restrictions as the default - since most other systems give no choice.
3. Failure to 'close' descriptions – since most other systems assume them closed by default.
4. Failure to anticipate the effects of domain and range constraints on reasoning.
5. Accidental creation of trivially satisfiable universal restrictions, which almost always represent errors, but may not cause logical errors in the early stages of development before there are any corresponding existential restrictions
6. Misunderstanding the difference between defined and primitive classes and the mechanics of converting one to the other

Note that, throughout this paper we will use the example of Pizzas which has proved a surprisingly rich field for tutorials. They are fun, concrete and real pizza menus are easily available. Defining a VegetarianPizza and other specific pizzas from the menu so that the correct classification is inferred by the reasoner turns out to be just the right amount of work to bring up the most common problems.

3. Addressing the problems

3.1 Making the easy thing the right thing – Existential quantifiers

The purpose of OWL is not just to create a concept hierarchy but to describe and define concepts. Therefore we want to 'build' some pizzas. OWL descriptions are built up of conjunctions (intersections) of conditions which can be either simple named classes or restrictions along property on the relations between classes.

Figure 1 gives a description of the concept MargheritaPizza⁴. One of the most common errors for newcomers to OWL is to use the universal (allValuesFrom) rather than existential (SomeValuesFrom).

The new interface greatly reduces the likelihood of this mistake by simply making the existential form the default. The user can make other choices, but the easy way is the right way. In addition, in the various wizards discussed later, the correct choice of existential quantifier is always made automatically.

```
OWL:
Class MargheritaPizza partial Pizza
restriction (hasTopping someValuesFrom
MozzarellaTopping)
restriction (hasTopping someValuesFrom
TomatoTopping)
```

Figure 1: Description of MargheritaPizza

3.2 Making it easy to do the right thing – Disjointness

The most important information commonly left out implicit is disjointness. In our example, having to make it explicit that meat, vegetables, fish, and cheese are all disjoint categories seems tedious and is easy to omit. Most other tools make a Unique Name Assumption so that this information *is* implicit in their representation. However, in OWL, if disjoint axiom are omitted nonsense toppings such as "vegetarian meat" are possible.

Our convention is that all primitive classes only have single parents and are disjoint. A primitive class is a class without a "complete" definition that helps form the basic "skeleton" of the ontology (these can be clearly seen as lighter nodes in Figure 5a). Therefore, a common operation when building an ontology is to make all classes disjoint from their siblings.

⁴ Margherita pizzas are listed in our Pizza Menu as having Mozzarella and Tomato toppings.

In previous editors, this required going to a separate axioms pane and entering the items individually. The first step in the new interface was therefore to provide a single button (see Figure 2) to make and all its siblings mutually disjoint.

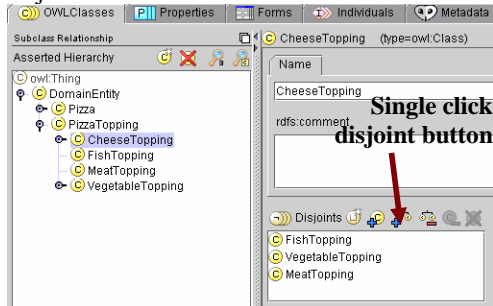


Figure 2: Disjointness widget – showing a single click being used to make Vegetable topping and all its siblings mutually disjoint.

However, even this is time consuming when a large hierarchy is being constructed. The next step was to provide a Wizard as shown in Figure 3 that managed the creation of a series of subclasses from a single starting point. The default in the wizard is that all the sibling classes are disjoint.

The Wizard has the added advantage that it drastically reduces the time needed to build the skeleton taxonomy for the ontology. Hence it is easy to get users to make this their usual means of creating lists of subclasses (Figure 3).

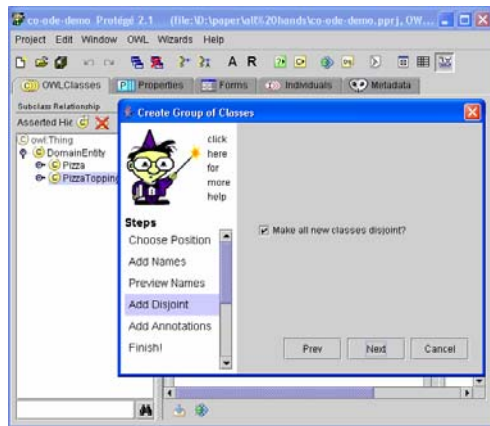


Figure 3: Create a group of classes using a Wizard

More recent advances in the classes tab allow the user to mark a class such that all of its subclasses are disjoint (this *only* applies to primitive classes). Using OWL annotations to store this information offers a flexible way of

adding different sorts of operations, with the benefit that they are stored when the file is saved. Experiments in using annotations in this way are showing how vital they will be for making the interface more powerful.

3.3 Making it easy to do everything – Value partitions

When creating pizza toppings or anything else, it is often possible to note several “constructs” which get used again and again. A collection of these common ontological patterns are being formalised as a W3C activity⁵. An example is where we need to create a set of values, and add properties for linking objects to those values. The detailed rationale is explained elsewhere [8] however the key problem for the user, and therefore the interface, is that the process involves a large series of operations. For example, to describe a quality of pizza toppings such as their “spiciness” one must:

- Create a subclass of ValuePartition to hold the values, *e.g.* SpicinessValuePartition
- Create the values as subclasses of the Value partition – *e.g.* Bland, Mild, Hot
- Make the values disjoint
- Make the values exhaustive by creating a ‘covering axiom’ – *i.e.* that the union Bland OR Mild OR Hot is logically equivalent to SpicinessValuePartition
- Create a property, *e.g.* hasSpiciness, and remember to make it functional (single valued)
- Set the range of the property to the value partition, *e.g.* SpicinessValuePartition

Because of the number of steps required to create this pattern, it is easy to omit a step by mistake. Combining all of these steps into a single wizard saves both errors and time. Also, because there are often several ways to model a similar construct, using a wizard offers consistency.

3.4 Making it easy to see what has been done – Untangling

The ontology in Figure 4 is typical of what students produce initially. We advocate a policy in which the primitives form a skeleton of pure trees - *i.e.* each primitive has exactly one primitive parent. When multiple

⁵ Ontology Engineering and Patterns Task Force
<http://www.w3.org/2001/sw/BestPractices/>

hierarchies appear in first drafts of the primitive hierarchy, as with SpicyTopping above, they must be untangled – *i.e.* the characteristics that differentiate the child concepts from all but one of the primitive parents must be made explicit. The overwhelming practical engineering advantage of this policy is that it makes the ontology more modular[8].

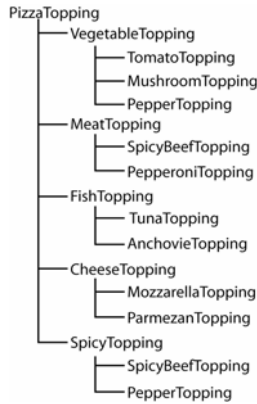


Figure 4: Tangled First Draft Pizza hierarchy

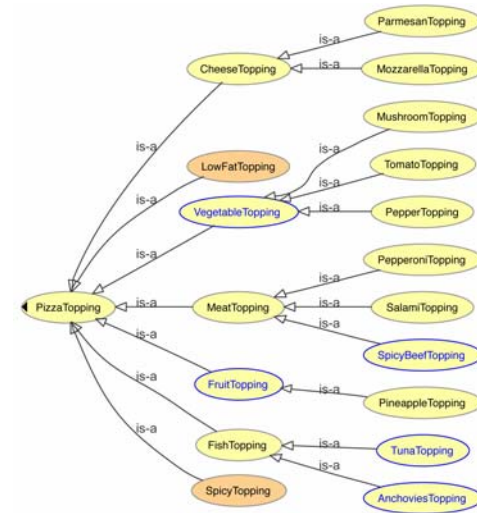
Using this methodology, a common migration path is for classes to first be created as primitives with only necessary conditions, and then later for some of those necessary conditions to be converted into definitions to untangle the hierarchy, for example by explicitly stating the spiciness of each ingredient. While this may seem tedious for a single abstraction such as spiciness, the effort to maintain the ontology when there are many such abstractions is drastically reduced. The effect is shown dramatically in the visualisations provided by the OwlViz package in Figure 5 in which just two abstractions – SpicyTopping and LowFatTopping have been added to the existing structure. Imagine the effect if there had been half dozen others as in a disease or drug ontology.

In the OWL syntax this appears simply to involve switching one keyword in the class axiom from partial to complete. However, there is an unexpected complication. It may be that not all of the necessary conditions on the class form part of its definition. For example, the ontology might have contained a restriction that SpicyToppings were not suitable for children. This statement is not part of the definition of SpicyTopping but a consequence of it. In OWL itself such restrictions must be removed from

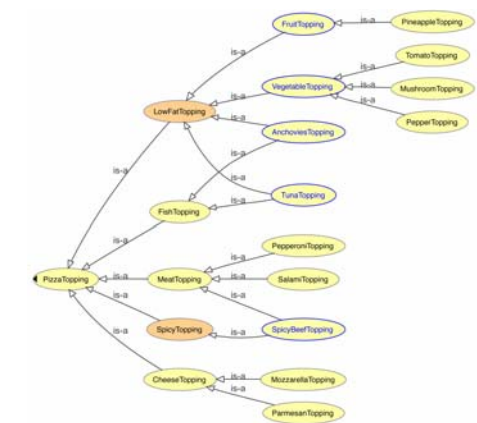
the class axiom and re-expressed in a separate subclassOf axiom⁶.

This has two effects. Firstly, it is tedious and error prone. More seriously, it spreads the information about SpicyTopping across two different constructs – the class and subclass axioms. In interfaces such as OilEd, which follow the syntax literally, this results in the information being on two separate tabs so that they cannot be viewed at the same time. This makes the ontology difficult to debug.

A major goal of the new interface has been to keep all information about a given class in a single view. This corresponds much more closely to the Frame view with which most are familiar.



(a) Initial hierarchy



(b) Classified hierarchy

Figure 5: Pizza hierarchy

⁶ This is a common feature of description logic style syntaxes such as Lisp.

This has been achieved by placing the definition (the necessary and sufficient conditions) and the merely necessary conditions into the same window (see Figure 6). Individual conditions can be moved between the two either by drag-and-drop or cut-and-paste.

This means that a primitive class is simply one that has no necessary and sufficient conditions; a defined class, one that has at least one set of such conditions⁷.

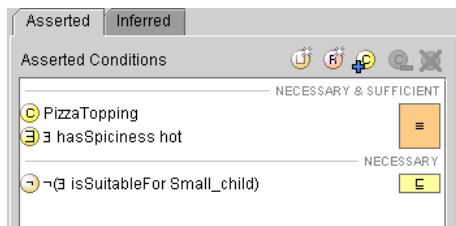


Figure 6: Two subpanes of the same window for defining and necessary restrictions.

3.5 Make it easy to find what is wrong – Time bombs and debugging aids

The use of a reasoner to assist in classification in OWL is one of its great advantages. However, it also means that writing in OWL is a kind of programming. If the reasoner makes unexpected inference; that a class is inconsistent or classified in the wrong place - or fails to make expected inferences e.g. fails to classify a MargheritaPizza under VegetarianPizza, the user must work out why and debug it. Several problems arise that make it difficult to track down errors:

- Property domain and range are not constraints but, instead, allow the reasoner to reclassify classes that do not use the property “correctly”.
- A single inconsistent class will cause any other class that uses it in the filler of an existential restriction to be inconsistent. Errors propagate, often causing the entire ontology to “turn red” from a single source error.
- There are certain “trivially satisfiable” restrictions that do not make definitions themselves inconsistent, but cause inconsistencies as soon as the user tries to use or specialise the class. These tend to be “time bombs” which

will cause problems later. For example, a VegetarianPizza defined as having only (VegetableToppings AND CheeseToppings) is consistent. Although there can be no topping which is both vegetable and cheese, there may be no requirement that a Pizza has any topping at all. The reasoner will not catch such errors until an additional restriction, usually on a subclass or superclass causes the reasoner to infer that there must be some topping. At this point, the affected class, and all classes related to it by existential restrictions, become inconsistent.

Three sorts of approach have been used to solve these problems.

- Checks before and during classification report on constructs which are likely to be errors – e.g. violations of domain constraints which cause a primitive to be reclassified. A user might have done this intentionally, but it is at least considered bad practice, and usually is actually an error.
- Additional information is provided after classification to test and search for the source of propagating errors.
- Checks for restrictions those are only trivially satisfiable (Figure 7). Most commonly these arise from the use of AND instead of OR (intersection instead of union), but they can arise in other ways.

Type	Source
⚠	hasTopping (MozzarellaTopping ∩ VegetableTo... Empty property value

Figure 7: Give warning for trivially satisfied restriction.

These debugging aids are still new, so experience is limited. They appear promising, but additional suggestions and ideas are invited.

⁷ Much debate has gone into the headings “necessary” and “necessary and sufficient.” Users reactions are still being sought – comments and alternatives welcome.

3.6 Applying the principles to a hard problem - Closure Axioms and Open World Reasoning

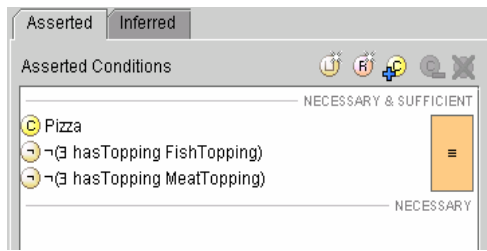


Figure 8: Definition of a VegetarianPizza.

The key task in our tutorials is to define a VegetarianPizza. A simple definition is shown in Figure 8. A VegetarianPizza is a Pizza that does not have any meat toppings and does not have any fish toppings. However, when we run the reasoner, MargheritaPizza (as defined in Figure 9) is not classified as a VegetarianPizza.

Why? Almost every new user falls into this trap. The answer is that unlike databases and logic programming, and virtually any other system prospective users may have met, OWL uses open world reasoning. In databases, logic programming or other ‘closed world’ systems, if something cannot be proved true, then it is assumed to be false in the one closed world of the reasoner. In OWL, something is only considered to be false when it is proved to be inconsistent in any possible world.

Any restaurant customer would assume that the menu entry meant ‘just mozzarella and tomato’, but there is nothing in the formal OWL definition to declare this so. In fact we can define a pizza with Mozzarella, Tomato and Salami, and the reasoner will classify it as a subclass of MargheritaPizza.

What is needed is a “closure axiom” that says that a MargheritaPizza has Mozzarella, Tomatoes and only those toppings as shown in Figure 10.

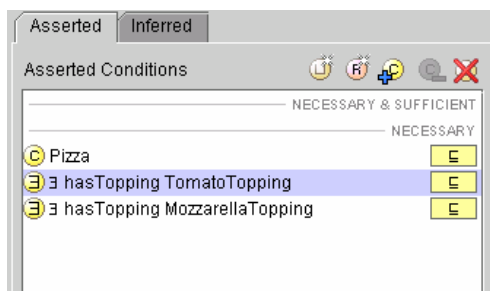


Figure 9: MargheritaPizza without Closure Axiom.

There are two problems in doing this:

- Many new users confuse the use of ‘and’ and ‘only’. The natural English representation of “Mozzarella and Tomatoes and only Mozzarella and Tomatoes” translates to three statements; two separate existential statements and one universal. Even users experienced in using query languages have been known to make this mistake. Anyone who has taught domain experts to use query languages will be unsurprised by the difficulties.
- The expression is potentially complicated and tedious to enter, even with a relatively optimised expression editor. Furthermore, in maintenance, each item has to be entered in two places.

The current solution is similar to the solution for disjoints; provide a single command to close a property for a class by generating the closure axiom automatically. The Protégé-OWL/CO-ODE/HyOntUse interface implemented such a command on the right mouse button for any existential restriction as shown in Figure 11. Using this mechanism it is a simple matter to close each kind of pizza.

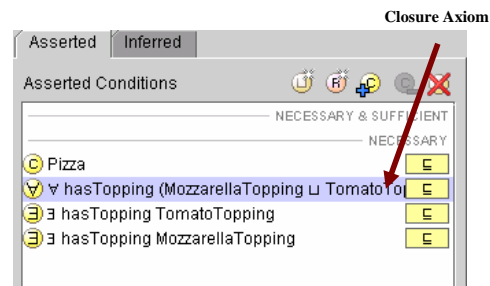


Figure 10: Closure Axiom for MargheritaPizza

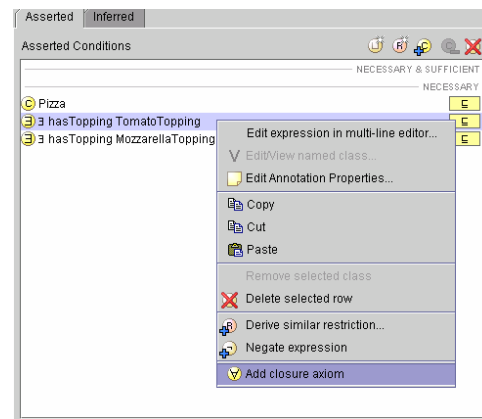


Figure 11: Adding Closure Axiom command

When closed, and the reasoner is re-run, the new classification of pizzas is correct.

4. Conclusion

The design of the interface for the new Protégé-OWL/CO-ODE/HyOntUse environment has been driven in large part by experience of errors made by users in tutorials, workshops, and practical experience. The intention is to provide several interfaces for different levels of user, but the current tools, aimed at logicians originally, have proved successful in initial experience, both in reducing errors and in speeding development. Further evaluations are taking place over the next few months with additional user groups.

All software is available at www.co-ode.org

Collaboration with user communities is an essential part of the Protégé-OWL/CO-ODE/HyOntUse programme. Users are invited to download the software, participate in the forums and contribute their views, experience, and ideas.

References

1. Guarino, N. and C. Welty. *Towards a methodology for ontology-based model engineering*. in *ECOOP-2000 Workshop on Model Engineering*. 2000. Cannes, France.
2. Noy, N.F. and D.L. McGuinness. *Ontology development 101: A guide to creating your first ontology*. 2001, Stanford Medical Informatics: Stanford CA.
3. Staab, S. and A. Maedche. *Ontology engineering beyond the modeling of concepts and relations*. in *ECAI 2000. 14th European Conference on Artificial Intelligence; Workshop on Applications of Ontologies and Problem-Solving Methods*. 2000.
4. Guarino, N., *Understanding, building and using ontologies*. *International Journal of Human Computer Studies*, 1997. **46**: p. 293-310.
5. Bechhofer, S., et al. *OilEd: a Reason-able Ontology Editor for the Semantic Web*. in *KI2001, Joint German/Austrian conference on Artificial Intelligence*. 2001. Vienna: Springer-Verlag.
6. Knublauch, H., O. Dameron, and M.A. Musen. *Weaving the biomedical semantic web with the protege owl plugin*. in *Proceedings of KR-Med-2004: International Workshop on Formal Biomedical Knowledge Representation*. 2004.
7. Rector, A., et al. *OWL Pizzas: Common errors & common patterns from practical experience of teaching OWL-DL*. in *European Knowledge Acquisition Workshop (EKAW-2004)*. 2004.
8. Rector, A. *Modularisation of Domain Ontologies Implemented in Description Logics and related formalisms including OWL*. in *Knowledge Capture 2003*. 2003. Sanibel Island, FL: ACM.