Modularity and OWL

Clark & Parsia

March 17, 2008

Contents

1 Introduction 3

2 Basic Modularity Approaches 5
  2.1 Against linguistic prescriptivism .................. 6
  2.2 Analytic modularity .............................. 11

3 Desiderata for Modular Importing 12

4 Modularization Algorithms 15
  4.1 Ad hoc algorithms .................................. 15
    4.1.1 PromptFactor algorithm ....................... 15
    4.1.2 Structure-based partitioning algorithm ........ 15
    4.1.3 GALEN segmentation algorithm ................ 16
  4.2 Formal algorithms ................................ 16
    4.2.1 CEL and MEX algorithms ....................... 17
    4.2.2 Cuenca Grau et al. (2006) algorithm .......... 17
    4.2.3 Locality-based algorithm ..................... 18

5 Summary and Outlook 19
  5.1 Outstanding Issues ................................ 20
  5.2 Future Directions ................................. 21

6 Prototype Tool Implementation 22
  6.1 Partial ontology importing ........................ 22
  6.2 Focused concept viewing .......................... 23

7 Acknowledgments 23

A Modular ontology development practices 24
Executive Summary

Web Ontology Language (OWL) provides a powerful knowledge representation language that has a clean and well-defined semantics based on Description Logics (DL). OWL is being used to develop many large-scale, complex ontologies like the NCI thesaurus, GALEN, etc. Such ontologies are developed by a team of knowledge engineers that collaborate together. However, OWL does not provide any means to support such a scenario, nor does it facilitate the partial reuse of ontologies in different contexts. As a result, ontology developers have to deal with large ontologies that get even larger, and, consequently, impossible to maintain upon integration with other ontologies.

Modular ontologies facilitate collaborative authoring and ontology reuse while improving the performance of reasoning. In this paper, we review the recent developments in modular ontology development techniques. We compare two basic approaches for modularity: (linguistically) prescriptive vs. analytic. Linguistic prescriptivism defines syntactic and semantic extensions to OWL so developers can define their modules a priori. We argue against such extensions for several reasons: there is no clear choice among proposals (E-connections, Distributed DLs, Package-based DLs), extensions require significant changes to the current infrastructure, and non-standard semantics proposed by these extensions are likely to confuse ontology developers.

We explain how analytic methods for modularity provide a viable alternative without major changes to the infrastructure. With the analytic approach ontologies are authored using standard OWL syntax and semantics but tools analyze the usage of external terms and extract a relevant module from the remote ontology on-the-fly. We identify the following conditions as necessary for this modular importing approach: module correctness, module completeness, module minimality, and import safety. We evaluate several modularization algorithms with respect to these criteria and examine which algorithms satisfy which conditions.

Out of all the modularization algorithms we investigate, the locality-based modularization is the most promising approach. Locality-based modules are proven to be correct and complete and empirically shown to approximate minimality better than other algorithms. However, we also discuss many open issues that need to be solved in order to use locality-based modularity approach in a large-scale collaborative ontology development environment. In addition, we describe our on-going efforts for implementing a Protégé4 plug-in to test some of the modularity ideas in practice and get a better understanding of the practical issues.
1 Introduction

OWL\textsuperscript{1} is an ontology language based on a fragment of First Order Logic (FOL). As a consequence, OWL inherits many strengths and weaknesses of FOL. For example, OWL is declarative with very well understood properties. This allows the operation of classification (i.e., determining whether a subclass relation holds between two terms) to work not just on named classes arranged in an explicit hierarchy, but on class definitions too. This permits bottom up terminology development \cite{23} and shifts the focus from “placing” terms correctly to defining them.

But First Order Logic was not developed with modern, large scale system development in mind. For example, it is relatively unstructured and even the idea of a class definition is implicit from a syntactic perspective. Furthermore, OWL allows one to make very powerful claims that can affect the whole ontology. Thus, to understand a term in an OWL ontology, one must be ready, potentially, to examine the entire ontology. As a corollary, if one makes a change to an OWL ontology, one must be ready, potentially, to examine the entire ontology for effects.

Representing and reasoning with multiple distinct, but linked OWL ontologies is crucial for building large and complex ontologies especially when they are authored collaboratively by a team of developers. A single monolithic ontology of large size causes problems both for humans and software tools. On the one hand, it is hard for ontology developers to understand and refine the defined terms; and, on the other hand, it is hard for software tools to process and reason with such ontologies.

Similar issues arise in related areas, e.g. software engineering. These problems are solved by building a large software program from smaller self-contained modules. For example, the programming language Java not only provides the standard Object-Oriented programming notion of class but also provides the notion of packages that contain classes belonging to the same category or providing similar functionality. One can import and use a single class from another package or even import single functions from other classes (using the static importing functionality introduced in Java 5). Modular software programs make maintenance and collaborative development much easier.

OWL provides some primitive means for modular ontology development. One of the historical problems in building ontologies from modular components is name collisions \cite{8}, e.g. combining two modules that use the same name to denote different terms. OWL, like RDF, solves this problem by using URIs to uniquely identify terms. The other feature OWL provides for combining ontologies is the \texttt{owl:imports} construct. This allows one to include, by reference, all the axioms contained in an external ontology in one’s local ontology. This means that everything in (the transitive closure of the imports of) the imported

\footnote{Throughout, we will use “OWL” to refer to the OWL-DL fragment of OWL specification. Without explicit qualification, we will use “OWL” to refer to the variant corresponding to the Description Logic $\text{SHOIN}$. We will use “OWL 1.1” refer to the revised version of OWL currently being developed by the W3C in the OWL WG.}
ontologies gets into the original ontology. Therefore, in practice, the only difference between using \texttt{owl:imports} and directly adding all the assertions in the imported ontology is that with \texttt{owl:imports}, the modeler can break an ontology into different documents. This is an important management feature but does not help with the understandability or scalability problems.

In summary, OWL does not provide any means to partially import and reuse parts of other ontologies. If we would like to use one term from a large ontology such as GALEN, we would end up importing the entire GALEN ontology. We could try to extract only the parts of GALEN that talk about the term we are interested in, but then we run into the problem of determining exactly what is or is not relevant. Aside from the non-local effects of axioms, we have the problem that GALEN is huge and complex. It would be a considerable amount of work to make a rough guess as to what is relevant and that guess could be very wrong. Even worse, an axiom we add to our own ontology could unintentionally change something in a remote part of GALEN. We can mitigate this, to some degree, by checking the classification and adding extra tests, but then we are once again faced with the fact that GALEN is difficult to reason with.

In the past few years there has been an explosion of work in the theory and practice of modular ontologies. A collection of language features, reasoning services, and methodological advances aim, with considerable success, at “taming” the power of OWL axioms without inhibiting them. Instead of arbitrary restrictions, modular analysis can allow ontology authors to determine the precise scope of influence of their changes. These techniques have been used to improve reasoning and to analyze the general structure of an ontology. Since they are also theoretically well founded, it is possible to understand precisely “what you get” from certain techniques and to compare them in a principled way.

There are different definitions describing what a module is. For now, we will provide a broad and informal definition that covers the use of the term in different contexts:

**Informal Module Definition.** A module of an ontology is a subset of the axioms in that ontology that has a certain coherent structure. Intuitively, the module should be self-contained in a way that enables us to reason with just that module, ignoring the rest of the axioms in the ontology.

We will use this informal definition to guide our intuition in comparing basic modularity approaches in Section 2, and as we progress to a better understanding of modules in Section 3 we will offer a more formal and crisp definition.

Once determined, modular structure can be helpful mainly in four scenarios:

1. **Collaborative editing.** Large and complex ontologies are developed not by a single person but by a team of developers, possibly coming from different organizations. Often each developer focuses on a subset of the ontology in her area of expertise and collaborates with other developers to relate the part she is working on with other parts of the ontology. We
not only want to facilitate concurrent editing of different modules but also guarantee that a definition in one module does not alter the meaning of terms in another module unintentionally.

2. **Partial ontology reuse.** Different subsets of large ontologies can be used outside the context of the original ontology [14, 21, 5]. The author of an ontology would like to publish such fragments to facilitate the reuse of those fragments by a wider audience. Users would like to extract such fragments to assist the development of their own ontologies.

3. **Ontology analysis and navigation.** The modular structure makes it easier for humans to analyze and understand the definitions in an ontology [29, 27]. First, modules help identify the domains modeled in an ontology. This makes it easier to visualize the ontology and navigate through the terms and their definition. Second, one can focus on certain modules that are relevant rather than looking at the whole ontology.

4. **Improved performance or scaling.** Importing large ontologies as a whole quickly causes the ontology size to become unmanageable by ontology editors and reasoners. For example, if we want to use one term from each of NCI thesaurus, GALEN, and FMA ontologies, then we would end up fully importing all three ontologies resulting in an ontology beyond the capabilities of any existing reasoner. On the other hand, the module for a single term would typically be very small compared to the whole ontology. Importing only the modules relevant for the terms we are interested in would decrease the overall size significantly, improving the performance of reasoners.

It is also important to mention another feature that has been considered desirable for modular ontologies: localized semantics for modules [6, 31]. Sometimes one may want to define a local view on the parts of an ontology that might not be globally true. In this setting, different modules would represent different contexts and reasoning with an ontology might give different results based on the chosen context. With contextual semantics, different modules can represent conflicting views of the same concepts without causing a global inconsistency because contextual information is local to the module and cannot affect other modules in any way. Localized semantics require considerable modifications to OWL both syntactically and semantically. In Section 2.1, we will examine such OWL extensions in more detail and explain why we think adopting such extensions are not suitable for large scale ontology development.

## 2 Basic Modularity Approaches

In general, a module of an ontology is a subset of the axioms that have a certain coherent structure, that is, that have certain properties. Intuitively, the module should be self-contained in a way that enables us to reason with just that module and ignore the rest of the ontology without losing inferences.
There are two basic approaches to exploiting modular structure:

1. **Prescriptive**: The user explicitly states what is in or outside of a module, i.e., the module boundaries; the role of the system is to respect those boundaries. Prescriptive approaches themselves come in two sorts:

   (a) linguistically prescriptive; that is, the language itself is changed, in syntax and semantics, so that violations of modularity are syntactic (or semantic) errors; the $\mathcal{E}$-connection approach is an example;

   (b) methodologically prescriptive; that is, we have some sort of validation mechanism, akin to current species validators, and we require users not to share an ontology without having coerced the ontology into a valid state.

   It is, of course, possible to combine these to some degree. For example, we might allow annotations to be added to certain axioms indicating that they are part of the same module. These would, as normal, be ignored by OWL reasoners, but we could have a separate tool which took the annotations as hints and provided warnings to the modeler when the hints were violated.

2. **Analytic**: Users write their ontologies in OWL as normal making full use of the language. There are a variety of system services which analyze the ontologies for modular structure, perhaps in response to queries from the user.

   Clearly, methodological prescriptivism relies on analytic services as they need to check whether some change has violated the hints. In fact, one could easily group methodological prescriptivism with methodological analysis in contrast with linguistic prescriptivism. In that respect, we will review some of the ontology development methodologies relevant for modularity in Appendix A.

### 2.1 Against linguistic prescriptivism

A key difference between linguistic prescriptivism (LP) and other approaches is that since LP typically radically changes the underlying language, it requires a radical change to the infrastructure as well. Since one is changing the syntax and semantics of OWL, OWL parsers and reasoners have to be revised as well. Furthermore, either one has to get these changes into the OWL standard, or one has to live with non-standard extensions as part of one’s ontology, which severely restricts reuse by other organizations and ties one to a customized, non-standard toolset.

Let us consider the case of $\mathcal{E}$-connections. $\mathcal{E}$-connections is a formalism that was designed primarily for combining different logics in a controlled way. In this way, it is very much in the spirit of data properties in OWL. OWL strictly segregates the set of properties into object and data properties. The objects of data properties are values drawn from a distinct set of individuals with an extensive predefined meaning. For example, a data property $\text{hasAge}$ may have
as its range the set of positive integers. The basic theory of integers (such as that one is less than two) is built into the logic; thus, the modeler does not have to include axioms enforcing the basic properties of integers. There are strict limitations on how a modeler can use these properties, for example, you cannot have inverse data properties (or, equivalently, properties on data values). By imposing these restrictions, reasoners can treat the data “universe” separately from the regular OWL universe. In fact, one can plug in different data theories (e.g., for strings) without changing the basic implementation of the reasoner. The data reasoner is a black box component of the combined (data and object) reasoner.

A key aspect of data properties is that the set of data values and the set of OWL individuals are disjoint and can interact in only very strictly controlled ways. \(\mathcal{E}\)-connections take a similar approach, except that:

1. instead of two disjoint sets of properties (data and object) we can have many distinct sets of properties (so called “link properties”);
2. instead of only two domains (i.e., object vs. data), we can have arbitrary numbers of them;
3. the language describing each domain can be OWL;
4. classes in each domain may be defined (in very restricted ways) in terms of any other domain (so we have possible two way dependencies).

As with data properties, the \(\mathcal{E}\)-connection extension to OWL introduces a fair bit of heavyweight syntax. We have a new family of properties called link properties which are associated with domains (component ontologies). Each domain can declare which foreign ontologies it “links” to. For each component, a modeler has to declare:

- the local classes, properties, instances, etc.; this generally just looks like normal OWL;
- a set of foreign ontologies and the link properties that connect the current component to the foreign one.

For example, suppose we are trying to build an ontology of bone diseases. We set up an \(\mathcal{E}\)-connection with two parts: a component about bones and a component about diseases. Since no bone is a disease, this separation is natural. However, we want to be able to define certain diseases as being associated with bones in general (e.g., bone cancer) or with specific types of bone. So we declare that `occursIn` is a link property in the `Disease` component which targets the `Bone` component ontology. Using the OWL syntax proposed in [15] the link property definition for `occursIn` would be written in the gene ontology as:

```owl

disease:occursIn rdf:type econn:LinkProperty ;
econn:foreignOntology BoneOntology ;
rdfs:domain disease:Disease ;
```

7
There are several things to note about this definition. First, we introduce a new built-in namespace for $\mathcal{E}$-connections denoted with the prefix econn\footnote{$\mathcal{E}$-connections has not gone through the W3C standardization process and does not have a standardized URI that can be used for the namespace of the newly introduced terms. The approach adopted in practice was to use the standard OWL namespace for $\mathcal{E}$-connection terms, which is against the W3C norms.} and several new terms (LinkProperty, ForeignClass, foreignOntology) defined under this namespace. The $\mathcal{E}$-connection terms are similar to the terms from RDF or OWL namespace; that is, they have a predefined semantics specified outside the ontology. Also, note that we explicitly specify the target of the link property as DiseaseOntology and type the class coming from the foreign ontology as econn:ForeignClass.

Continuing the example, if we have Tibia defined as a subclass of Bone in the bone component and Tibial-Adamantinoma defined as a subclass of Disease in the disease ontology, we can define the association between them as follows:


    bone:Tibia rdf:type econn:ForeignClass ; econn:foreignOntology BoneOntology .

The syntactic extensions of $\mathcal{E}$-connections give ontology developers precise control over how ontologies interact with each other by explicitly specifying which foreign concepts are used and how they are linked to the source ontology. However, we can easily get into trouble with this approach. For example, suppose we want to enrich our ontology with a theory of cells: Where should we define the class Cell? Perhaps including it with Bone makes the most sense since we can then define certain bone cells as being cancerous (but not by being a subset of a class called CancerousCell). Each substantively new concept requires significant analysis on the part of the modeler before she writes any axioms. The modeler must make significant, high level decisions very early in the process. These decisions strongly shape the subsequent modeling.

$\mathcal{E}$-connections (and $\mathcal{E}$-connection derived segmentation algorithms such as [16, 17]) are biased to what we might call “vertical” separation of concerns: They work best when there are parallel subject domains which are related by properties with the subject in one domain and the object in another. They do not help at all in the following case: two people or groups are trying to refine sibling classes without interfering with each other.

There are other modular ontology languages that tackle this specific problem of specifying directional (“view dependent”) subsumption relationships between
concepts coming from different ontologies. The Distributed Description Logics (DDL) [6] formalism introduces the notion of “bridge rules” to define subsumption, equivalence and disjointness between concepts of different ontologies. Context OWL (C-OWL) [7] extends OWL syntax and semantics with DDL bridge rules.

Let’s examine the extensions DDL (and C-OWL) provides by looking at the example of combining GALEN and UMLS ontologies described in [7]:

Both GALEN and UMLS ontologies describe the notion of “substances” that are involved in physiological processes. However, the actual notion of substance as defined in GALEN is not exactly equivalent to the notion of substance in UMLS, because it also contains some notions that are found under anatomical structures in UMLS. GALEN also contains the notion of a generalized substance which is a notion of substance that subsumes substances in a physical sense making it more general than the notion of substance in UMLS.

These relationships between GALEN and UMLS concepts can be expressed with the following two bridge rules [7]: \texttt{golen:GeneralisedSubstance} is a superclass of \texttt{umls:Substance} and \texttt{golen:GeneralisedSubstance} is a subclass of the union of classes \texttt{umls:Substance} and \texttt{umls:AnatomicalStructure}. In C-OWL syntax, a superclass mapping is done with an \texttt{cowl:Onto} rule and a subclass mapping is specified with an \texttt{cowl:Into} rule. So the above concept mappings would be written as follows:

\begin{verbatim}
GALEN-UMLS-Mapping rdf:type cowl:Mapping ;
cowl:sourceOntology GALEN ;
cowl:targetOntology UMLS ;
cowl:bridgeRule [
  rdf:type cowl:Onto ;
cowl:source golen:GeneralisedSubstance ;
cowl:target umls:Substance ] ;
cowl:bridgeRule [
  rdf:type cowl:Into ;
cowl:source golen:Substance ;
cowl:target [
    rdf:type owl:Class ;
    owl:unionOf (umls:Substance
\end{verbatim}

Note that, C-OWL also introduces another non-standard namespace \texttt{cowl} and a tool needs to know about C-OWL syntax and semantics to process these bridge rules properly. Further, C-OWL allows concept subsumption that \(\mathcal{E}\)-connections does not provide; but, in contrast, C-OWL cannot reuse foreign concepts in restriction as in \(\mathcal{E}\)-connections.

There is yet another LP approach called Package-based Description Logics (P-DL) [3] which tries to overcome the expressive limitations in \(\mathcal{E}\)-connections.
and DDLs by allowing both inter-module concept subsumptions and foreign concepts in restrictions. However, as in other LP approaches, P-DL defines another non-standard localized semantics; and, furthermore, reasoning support is provided only for the very inexpressive DL $\mathcal{ALC}$ which lacks various important features of OWL, e.g. property subsumption, transitivity and cardinality restrictions.

In recent surveys [14, 31], we see as many as five families of modular extensions to OWL. While some can be reduced or encoded in others, some cannot easily be intermapped. These are clearly a small subset of the possible formalisms that could be explored. Thus we run into several major problems with the LP approach:

- Each LP approach extends OWL syntax and semantics significantly which means any option would require considerable upgrades to the infrastructure including tools (parsers, editors, reasoners, etc.) and standards.\footnote{We strongly believe that there is no way that the current working group would consider such radical changes to OWL 1.1} For example, the $\mathcal{E}$-connection support was implemented in the ontology editor Swoop and reasoner Pellet with modifications to the parser and model of OWL API. However, over the years maintenance of this non-standard extensions has become so problematic that this support has been removed in the recent releases of these tools.

- If NCI is going to commit to the huge effort of upgrading the infrastructure, then we need to be sure that the formalism we commit to serves current and future purposes well. There is no apparent consensus in the literature on one LP approach. All use of these formalisms seems to be driven by researchers in academia rather than having caught the imagination and solved the problems of the user base.

- We also share the sentiment expressed in [14] that modular semantics are likely to confuse the user. It is already a challenging task for users to fully grasp the semantics of OWL (Open World Assumption, no Unique Names, inferencing in contrast to integrity constraints, etc.). It would be much harder for users to understand and correctly use the non-standard localized semantics provided by LP approaches. It is also non-trivial to determine which formal properties should be expected from them and to establish their relationship with the conventional OWL semantics.

The authors of [14] do point out that linguistic prescriptivism can have some compelling advantages, in particular, it tends to offer more control and tends to be more intention revealing. Thus, for example, with $\mathcal{E}$-connections, the modeler can force certain terms together even if their axioms do not, logically speaking, bind the use of the terms. Similarly, an $\mathcal{E}$-connected ontology has an explicit set of components which reflect a way of dividing up the overall subject matter. But this also proves to be a disadvantage in cases where the ontology author cannot anticipate all the different ways her ontology will be reused. In
certain application domains, a different kind of modularization of the terms can be more useful but cannot be achieved with a priori prescribed modules.

### 2.2 Analytic modularity

Given the high expense of changing the base language, it seems clear that linguistic prescriptivism, at least in the sense of changing the base formalism, is a non-starter. Lighter-weight approaches are less risky in general and have the great advantage of being easier to adapt to changing needs and our changing methodological understanding.

In an analytic approach, users author their ontologies in OWL with the standard syntax and semantics including the `owl:imports` statement. Then ontology tools analyze these descriptions and provide services to facilitate modular design.

The most important service is examining how external terms are used in the local ontology and dynamically extracting a relevant module from the foreign ontology. Intuitively, a module of the foreign ontology is defined to be the set of axioms that are relevant for the terms used in the local ontology. In Section 3, we will discuss this issue in more detail and provide other properties one can expect from modules.

Dynamically extracted modules can then be used in ontology viewers, editors, and reasoners. An editor would show only the concepts and axioms included in the module and reasoners would only process the axioms from that module. This dynamic module extraction technique, which we will call modular importing, allows us to partially reuse any terms from an ontology in a scalable manner.

The idea of modular importing has been widely investigated in recent years [10, 19, 3, 5, 25, 4]. Some of these approaches (e.g. [25, 4]) investigate the notion of “semantic imports” and are in fact closer to the LP approach because they are based on non-standard semantics (not surprisingly [25] has roots in DDL and [4] has roots in P-DL). They suffer from the same issues we discussed in the previous section. Other approaches [19, 5], on the other hand, are based on module extraction algorithms that do not require any change to the OWL semantics.

One common feature in modular importing proposals of [19, 5] is to augment the import statements with explicit specification of which terms (concept or property) should be reimported. Explicit specification of imported terms is not really necessary because tools can easily discover such information automatically by examining the axioms in the local ontology and collecting external terms used. So the choice between explicit specification and automatic detection reduces to the preference of ontology developers. In our prototype implementation described in Section 6 we chose the latter approach since it is less intrusive.

The main question in analytic modularity approach is what kind of modularization algorithm to use to extract the relevant parts of an ontology. There are various alternatives [13, 24, 28, 29, 21] one can choose from which would result in (sometimes significantly) different modules. At first, this might seem like the
same situation as in the LP approach; lots of choices with no apparent winner. But there is a very important difference: the ontologies developed are exactly the same in all cases. Modular importing is not a feature of the ontology but a feature of the tool that processes the ontology. So, in principle, different tools may use different modularization algorithms (provided that those algorithms satisfy some conditions we will discuss shortly). Furthermore, a tool can simply use the standard OWL importing mechanism always importing entire ontologies and everything would still work (with the obvious difference that such tools would suffer from the scalability problems we mentioned earlier).

For modular importing to work in practice, there are some conditions that a module should satisfy. We would want the modules to be logically complete so we don’t miss entailments that would have been inferred with regular importing. In the following section, we will examine such conditions in more detail.

3 Desiderata for Modular Importing

The semantics of OWL is defined in terms of reasoning with respect to full imports closure of an ontology. Every reasoning service such as concept satisfiability, concept subsumption, etc. should be done by considering all the axioms in the imports closure. With modular importing we only want to consider a subset of the axioms that is relevant for the terms we are working. As we discussed earlier, it is very hard to determine the modules of an OWL ontology. Thus, we need to use a formal methodology to decide what is a module in an ontology.

The main properties that a module need to satisfy have been studied and identified in the modularity literature [13, 14, 21, 22, 4, 29]. Not everybody agrees on all the properties proposed in the literature; for example, we have argued, as have the authors of [14], against the non-standard semantics that are adopted by [25, 4]. However, there is almost a consensus on the main properties of modules:

(D1) **Module Correctness** A module of an ontology should contain only information that is present in the original ontology. This means any inference deduced from the module should be deduced from the original ontology. This objective is easy to fulfill if the module is a syntactic subset of the original ontology with respect to the asserted axioms it contains.

(D2) **Module Completeness** A module of an ontology should contain all the information from the original ontology that is relevant for the terms that are reused in the importing ontology. This goal makes sure that the ontology developer will not see any differences in the inferences between importing the module and importing the whole ontology.

(D3) **Module Minimality** A module of an ontology should be as small as possible. The simplest module definition for an ontology that satisfies correctness and completeness is the ontology itself. Obviously such modules are not
useful and do not help with any of the use cases we identified earlier in Section 1.

It is straightforward to formally define the correctness and completeness criteria using OWL semantics specification as done in [13] and [21]. As we mention above, correctness is a fundamental property that simply follows from the monotonicity of OWL when the module is computed as a subset of the axioms in the ontology. All the modularity algorithms that we will examine in Section 4 trivially satisfy this property.

The completeness of a module is not as straightforward as correctness but a formal definition for it has been established recently based on the notion of conservative extensions [9]. An ontology is defined to be a conservative extension of another ontology with respect to a set of terms if the meaning of those terms is not changed in the extension ontology. Then, we say that subset of an ontology is a module of that ontology if the ontology is a conservative extension of the subset. That is, if the the meaning of the terms in the module are not changed outside the module then the module is considered to be complete for those terms.

Based on the formal definition of correctness and completeness, we can easily prove or disprove that the modules generated by a certain algorithm are correct and/or complete. There are no degrees of correctness or completeness, i.e. a module is either correct (respectively complete) or not.

In contrast to the crisp notions of correctness and completeness, the minimality of modules can vary and in most cases the best we can do is to approximate the minimal module (see Section 4.2 for more discussion). For this reason, the sensible way to compare two different modularization algorithms with respect to minimality is empirical analysis.

The completeness and minimality conditions are two competing goals: we need to include more axioms in the module to ensure completeness which increases the size of the module and works against minimality. We believe completeness is an important property that should always be respected for the purposes of reasoning. However, for certain tasks like navigation and visualization, minimality can be more important than completeness, e.g. if the goal is to provide a small fragment of the ontology that will help the user understand the definition of term in an ontology. The minimality of module in this context is highly subjective and hard to quantify or formalize. We will revisit this issue in Section 5.1.

**Formal Module Definition** A module of an ontology for a given set of terms is a minimal subset of the axioms in that ontology where the entailments inferred from the module are correct and complete with respect to the entailments inferred from the whole ontology involving those terms.

There is another property that has been identified in the modularity literature; however, this, unlike the first three properties, is not a property of the imported module but a property of the importing ontology.
Import Safety The meaning of the imported terms should not be changed in the importing ontology. In other words, any relation between two imported terms such as equivalence or subsumption that can be inferred from the importing ontology should also be inferred from the imported ontology itself. Safe reuse means that the importing ontology may define new subclasses or superclasses of imported terms but cannot define a subclass relation between two imported terms.

Note that, the safety condition is also directly related to the conservative extension notion. An importing ontology is safe for the imported ontology if their union is a conservative extension of the imported ontology. Safety ensures that by importing an ontology one will not unintentionally change the meaning of terms in that ontology. In OWL, there is no mechanism to prevent or detect such cases. This means a user can import some terms from a foundational upper ontology, create some axioms using the imported terms and add new subsumption relations between the imported terms. Affecting the upper ontology like this is not desirable for several reasons. First, it might be the case that this change does not fully comply with the original modeling in the upper ontology and is in some sense wrong. Second, it might be the case that the upper ontology was missing that added relation, but adding this to only the local importing ontology means none of the other users of the upper ontology will see the added relation. In this case, it would have be better to add this missing relation into the upper ontology directly.

Ensuring safety also improves collaborative editing of ontologies. When an ontology is partitioned (possibly automatically) into modules, and there are different authors responsible of developing each module, we do not want the developers of one module to change the meaning of terms defined in the another module. Such effects violate the separation-of-duty constraints one would like to establish on the development of modules and should better be handled by explicit communication between developers.

It is also important to recognize that in some cases an ontology developer intentionally would like to change the meaning of imported terms. For the upper ontology example, it might be the case that the terms of interest are not defined in sufficient detail for the purposes of the importing ontology and it is not possible to update the upper ontology. Or there could be some additional relations that are true for the application domain of the importing ontology but that do not globally hold. In such cases, one would disregard safety and change the meaning of imported terms.

It is also possible to demand the safety of some imported terms but not others. If the ontology developer knows that she will change the meaning of some imported classes, then safety condition might be enforced only for other terms that the developer does not wish to change. Parametric safety enforcement provides the flexibility of changing the meaning of terms when required, while preventing the unintentional changes to other terms. Unfortunately, such nuanced safety notions have not been studied in the literature.
4 Modularization Algorithms

As mentioned earlier, there are many different algorithms described in the literature which are designed to extract the modules of an ontology. Throughout this section, we will review the most prominent modularization algorithms mentioned in the literature and evaluate their fitness based on the desiderata we specified in the previous section.

It is important to note that some of these algorithms generate modules simply as a set of concepts rather than a set of axioms as in the definition we have been using. The straightforward way to construct a set of axioms from a given set of concepts is to simply include all the axioms that mention those concepts.

4.1 Ad hoc algorithms

A common approach in modularization algorithms is traversing the axioms in the ontology and using heuristics to determine which axioms should be in a module. These modularization procedures do not attempt to formally specify the intended outputs of the procedures, but rather argue what should or should not be in the modules based on intuitive notions. In particular, they do not take the semantics of the ontology languages into account. Therefore, such algorithms do not guarantee the completeness of the extracted module. Because of these reasons, we do not recommend such ad hoc algorithms.

4.1.1 PromptFactor algorithm

Perhaps the most commonly used modularity algorithm is the PromptFactor tool \[24\] that is available as a Protégé3 plug-in. The PromptFactor algorithm extracts a fragment of an ontology based on a given signature. The algorithm starts by finding the axioms that mention the terms in the given signature and expands the signature with other terms mentioned in those axioms. The expansion continues until a fixpoint is reached. The correctness of the modules generated by the algorithm is obvious (since an extracted module is a subset of syntactic axioms in the ontology) but the completeness of modules cannot be guaranteed. It has been shown in \[13\] that PromptFactor modules are not complete in the presence of unsatisfiable concepts or in the presence of enumerations. It is possible, though not straightforward, to prove the completeness of the algorithm if the ontology meets certain conditions but this has not been done in the literature. Furthermore, empirical evaluation of PromptFactor \[13\] shows that the size of the modules computed by this algorithm can be two orders of magnitude larger than the size of the modules generated by algorithms that can guarantee completeness.

4.1.2 Structure-based partitioning algorithm

Stuckenschmidt and Klein present a method of partitioning the class hierarchy into modules \[29\]. They exploit the structure of the hierarchy and constraints
on properties’ domains and ranges to iteratively break the ontology up into
dynamically sized modules. This method does not take OWL restrictions into
account at all and instead relies on the globally asserted domain and range
constraints.

Structure-based partitioning is primarily meant for breaking an ontology into
broad packages or modules so that it can be more easily browsed and visualized.
Thus, the modules it generates fail to satisfy the completeness criteria.

4.1.3 GALEN segmentation algorithm

Seidenberg and Rector [28] describes another ad hoc modularity algorithm which
was primarily designed to extract modules from the GALEN ontology but can be
applied to arbitrary OWL ontologies. The algorithm is similar to the structure-
based partitioning algorithm but uses different kinds of structural analysis and
heuristics, specifically the algorithm takes OWL restrictions into account (at
least syntactically).

The description of the procedure in [28] is very high-level. The authors
discuss which classes and properties should be included in the segment and
which should not based on intuitive “usefulness” criteria. In particular, they
argue a module should contain all sub and superclasses of the input class plus all
the classes that are linked via existential restrictions from the input class. The
superclasses of the classes included via links are also put into the module but
their subclasses are not. For any included class, the algorithm also inspects their
restrictions, intersection, union, and equivalent classes and pulls all the classes
and properties used in those descriptions. Several (semi-automatic) filtering
methods are also described to prune some of the terms from the module to keep
the module size small.

The behavior of the algorithm is justified based on intuitions and/or struc-
tural analysis rather than semantics. As a result, it is easy to come up with
examples that demonstrate the incompleteness of modules generated [10].

4.2 Formal algorithms

Formal modularization algorithms adopt a principled way of extracting modules
that are correct and complete. As we explained in Section 3, the completeness
of modules are defined in terms of conservative extensions. Unfortunately, it
has been shown that determining if an ontology is a conservative extension of
another one is undecidable for OWL-DL and becomes decidable only for less
expressive fragments [13, 21, 22]. There are then two alternatives to tackle
the problem: focusing on simple inexpressive DLs or approximate the solution by
finding sufficient conditions that will ensure completeness. In practical terms,
approximating means that we might end up with larger modules than theo-
retically needed but the module generated will be provably complete. Both of
these approaches have been tried in the literature and we will now review these
algorithms.
4.2.1 CEL and MEX algorithms

There are two different modularization algorithms geared toward tractable fragments of OWL, namely the \( \mathcal{EL} \) fragment \([1]\). The \( \mathcal{EL} \) family of DLs restrict the expressivity of OWL to gain tractability, but it is still expressive enough to formulate some of the important life science ontologies like GO and SNOMED-CT. Reasoning in \( \mathcal{EL}^{++} \) (the most expressive language in the \( \mathcal{EL} \) family) is polynomial time and has been implemented in the CEL reasoner \([2]\).

The CEL reasoner, in addition to standard reasoning services like classification, provides a modularization service based on connected reachability \([30]\). The nodes in the reachability graph are labeled with concepts from the ontology and edges are labeled with axioms. Intuitively, a concept \( A \) is reachable from another concept \( B \) if \( B \) syntactically refers to \( A \) either directly or indirectly via the axioms in the ontology. The reachability between two concepts suggests a potential subsumption relationship between them. The module of a concept includes the concepts in its connected component and the axioms in the label of nodes. This axiom is guaranteed to be generate logically complete modules for \( \mathcal{EL}^{+} \) ontologies.

MEX is another module extraction algorithm for \( \mathcal{EL} \) ontologies that has been proposed recently \([20]\). The algorithm generates (uniquely determined) minimal modules but only for acyclic \( \mathcal{EL} \) ontologies (i.e. there can be no cyclic class definitions). For acyclic \( \mathcal{EL} \) ontologies MEX generates smaller modules than CEL reasoner and any other more generic module extraction algorithm. However, both algorithms have very minimal applicability due to being limited to \( \mathcal{EL} \) fragment. Many ontologies including NCI thesaurus and GALEN are beyond the expressivity of \( \mathcal{EL} \).

4.2.2 Cuenca Grau et al. (2006) algorithm

Cuenca Grau et al. (2006) \([17]\) describes a modularity algorithm that satisfies the correctness and completeness criteria of Section 3. The algorithm also imposes an additional requirement on modules, namely, that the module should entail all the subsumptions in the original ontology between atomic concepts in the module and other concepts in the ontology. This additional restriction makes it possible to devise a quadratic time modularization algorithm, but also means it is a rougher approximation to the minimality and the modules generated will be larger.

The algorithm also requires the original ontology meet certain conditions in order to be considered safe for modularization.\(^4\) For example, the algorithm cannot extract modules from an ontology that contains non-safe axioms, e.g. an axiom that declares \( \text{owl:Thing} \) to be a subclass of another (possibly complex) class is considered to be non-safe.

The main idea of the algorithm is to generate a partitioning of the input ontology represented as a directed labeled graph (the partitioning graph). The

\(^4\)Note that, the term safety in this context is not same as the safety condition we described in Section 3
nodes in the graph are labeled with the axioms of the ontology and the edges are labeled with properties. Each concept is also assigned to a node in the graph. The algorithm starts with a graph of single node that contains all the axioms, then moves one of the terms to a newly created node along with axioms associated with it. In the end, the module for a term is computed by finding the connected component of its node and including all the axioms in the label of that component’s nodes.

This algorithm is completely syntactic, i.e. no reasoner is used during module extraction, but it is proven to guarantee logical completeness. The major drawback is the limitation to safe ontologies and the relatively large modules computed by the algorithm [13].

4.2.3 Locality-based algorithm

A more promising and tractable modularization algorithm is based on the notion of locality that was proposed in [11, 10] and further refined in [13]. These ideas have been used to support modular and safe imports of ontologies [19].

The locality notion is another approximation to the conservative extensions to make the problem decidable. Intuitively, an axiom is local for a set of terms if the axiom does not change the meaning of those terms in a certain way. There are as many as six different types of locality conditions identified [13] and each locality type has a different definition of what it means to change the meaning of a term. The most commonly used locality types are ⊤-locality (top-locality) and ⊥-locality (bottom-locality). An axiom is ⊤-local for a class if it does not define a new subclass for that concept and ⊥-local if it does not define a new superclass for that concept.

Different locality types lead to different types of modules. For example, the modules extracted for a set of classes using ⊥-locality are called upper modules and are guaranteed to contain all the superclasses of those classes. Similarly, using ⊤-locality yields lower modules which are guaranteed to contain all the subclasses.

One can decide which of the upper or lower modules to use based on what the importing module is supposed to do. For example, if we are importing a foundational upper ontology, then we would most likely refine the concepts defined in the upper ontology, i.e. define more specific subclasses tailored for our specific domain. In such a case, we would like to use upper modules so we can get the complete set of superclasses for the concepts we are defining. In other cases, we might want to generalize the concepts of the imported ontology, in which case we want to use lower modules.

The safety of imported terms is guaranteed if the importing ontology either only refines or only generalizes the imported terms. Note that this is an all-or-nothing approach which might be very restrictive in many cases. For example, if we are reusing two completely unrelated concepts from the imported ontology by refining one and generalizing the other, this approach would conclude that import safety is violated even though there could be no interactions between those two terms.
There are two different methods developed to extract modules based on locality: semantic locality and syntactic locality. Semantic locality transforms axioms in a certain way (again the actual transformation depends on the locality type used [13]) and then checks for tautologies using a standard DL reasoner. Checking for tautologies in DLs is, theoretically, a difficult problem but we are considering each axiom in isolation so the size of the ontology does not matter and the standard optimizations in DL reasoners can effectively find tautologies. Syntactic locality, on the other hand, does not require a reasoner and can be implemented in polynomial time using purely syntactic checks. The caveat is syntactic locality is an approximation to semantic locality (which itself was an approximation) so an axiom can be syntactically non-local but semantically local. What this means, as in other cases of approximations, is that modules extracted with syntactic locality will be larger compared to modules extracted with semantic locality.

5 Summary and Outlook

In this paper, we have reviewed the recent developments in modular ontology development. Modular ontologies facilitate collaborative ontology authoring and partial ontology reuse while improving the performance of tasks like reasoning. Modular structure also makes it easier for developers to navigate and understand the contents of large ontologies.

We have first compared two basic approaches for modularity: (linguistically) prescriptive vs. analytic. Linguistic prescriptivism defines syntactic and semantic extensions to OWL language to let developers define their modules a priori. We argued against language extensions for several reasons. First, there is no clear choice among possible language extensions (E-connections, DDL, P-DL). Second, adopting these extensions would require significant changes to the current infrastructure including updates to tools (parsers, editors, reasoners, etc.) and standards. Third, the non-standard semantics proposed by these extensions are likely to confuse ontology developers and make it harder to learn and understand the new semantics.

We explained how analytic methods for modularity provide a viable alternative without major changes to the underlying infrastructure. With the analytic approach ontologies are authored using standard OWL syntax and semantics but tools analyze the usage of imports and imported terms to extract modules on-the-fly. We argued that dynamic computation of modules provides flexibility to allow different parts of an ontology to be modularly imported in different contexts (possibly in ways that are not anticipated by the original author at design time).

We identified several conditions that need to be satisfied by modules so that modular imports can be used instead of complete imports. These conditions are module correctness, module completeness, module minimality, and import safety. We evaluated several modularization algorithms with respect to these criteria and examined which algorithms satisfy which conditions.
Module correctness is trivial to achieve and was satisfied by all the algorithms we evaluated. Module completeness, on the other hand, is harder to accomplish and we have seen that many ad hoc algorithms fail on that criteria. Computing minimal modules turns out to be undecidable except for relatively inexpressive DLs [21, 22] but good approximation algorithms have been developed. Lastly, we argued that import safety is an important notion that need to be respected in certain cases but it is also perfectly reasonable to ignore safety in some other use cases.

Out of all the modularization algorithms we looked at, the locality-based approach is the most promising one. Locality-based modules are proven to be correct and complete. The empirical analysis [13] show that locality-based modularization approximates minimal modules much better than other algorithms. It is also possible to ensure imports safety using locality-based modules, though safety restriction limits its applicability considerably.

There are, however, many issues that need to be solved in order to use locality-based modular importing approach in a large-scale collaborative ontology development environment. We will discuss these outstanding issue in the following subsection. We are also in the process of implementing a Protégé plug-in to test some of these ideas in practice and get better understanding of the practical issues. We will describe the current state of the implementation in Section 6 and provide a more detailed report once the implementation is finished.

5.1 Outstanding Issues

Our analysis of modularization algorithms focused on formal properties that the resulting modules should satisfy. For example, we argued for the necessity of module completeness so that the importing ontology would not miss any definitions and entailments that are relevant for the imported terms. But logical completeness do not always result in coherent and understandable modules. The different types of modules one gets with locality-based modularity, i.e. upper and lower modules, is an example of this.

Let’s illustrate the last point with an example. Suppose we are refining an imported class C in our ontology by defining its subclass D. Then we would extract and import the upper module for C which would be guaranteed to contain all its superclasses but might not include (any of) its subclasses. That would mean that we would not even see any of the sibling classes of D. From a logical entailment point of view, this is acceptable because without any more axioms, we cannot infer any relation between D and any other subclass of C. But from the modeler’s point of view, this is not very satisfactory as sibling classes of D would most likely be considered relevant.

The unintuitiveness of such modules from the modeling perspective suggest that there is more work in the area of extracting comprehensive and understandable modules. Logical completeness should be considered the minimal set of axioms we will consider, but this module can be extended further with heuristics or user-defined rules that may depend on the application domain. This
hybrid approach would both have the strength of rigorous, formal approach and capture the intuitive results that the ad hoc algorithms are aiming for. Unfortunately, intuitiveness is a very subjective notion that cannot be quantified or tested objectively, so such extensions would mostly rely on user studies and usability tests.

The safety requirements imposed by the locality-based modularity approach are very restrictive, as we pointed out in Section 4.2.3. The importing ontology should commit to one of refinement (subclass definitions) or generalization (superclass definition) for all the imported classes. We cannot refine one class and generalize another one since that violates the safety condition. Furthermore, defining one of the local classes equivalent to an imported class will be considered unsafe automatically because equivalence implies both sub and superclass relation. Defining a synonym for an imported term is a very common use case and would not be considered unsafe if we were using the conservative extension definition for safety. However, locality, being an approximation of conservative extensions, treats this usage as unsafe.

The restrictiveness of locality-based safety condition limits its applicability in a number of modeling scenarios. In those use cases, the user is forced to ignore the safety violations, since there is no actual safety violation, which might later cause a real safety violation to be missed. Furthermore, it is not straightforward how to decouple the import safety requirements from module completeness so one can extract correct and complete modules even when the imported terms are used unsafely.

There are also some outstanding issues regarding the lack of class declarations in OWL. It is not always possible to analyze an ontology and determine which classes are defined in the ontology vs. which classes are imported. This is easy to resolve when there are not any cyclic import statements but gets much harder, if not impossible, in the presence of cyclic imports. OWL 1.1 introduces the notion of class declarations to solve this problem, but it is not clear if this feature will be in the final version of OWL 1.1 specification. Even if this feature is included in the OWL 1.1 specification, what to do with existing OWL ontologies with cyclic imports remains an open question.

5.2 Future Directions

We are currently looking at relaxing some of the restrictions related to locality-based modularity approach as explained in the previous section. Relaxing the import safety conditions and decoupling import safety from module completeness are two major issues we are investigating. Improving the efficiency of the implementation is also crucial to update the imported modules in dynamically in response dynamically to changes to the ontology.

We are also looking at reducing the size of extracted modules. Using semantic locality, instead of syntactic locality, will possibly give better results but an empirical evaluation is needed to measure how much the module sizes change in typical cases. Another approach we are considering is combining the locality-based modularity with the modularity algorithms developed for $\mathcal{EL}$ ontologies.
(see Section 4.2.1) which compute much smaller modules. Even though many ontologies do not fit into the expressivity of $\mathcal{EL}$ family, it is possible that the modules we extract with locality will fit into $\mathcal{EL}$ since they contain significantly fewer axioms. Therefore, we can have a two-stage modularization algorithm where we first extract a coarse module with locality and then apply the $\mathcal{EL}$ modularity algorithm to get a fine-grained module.

6 Prototype Tool Implementation

We have started building a prototype tool to support modular ontology development. Our proof-of-concept implementation is a Protégé plug-in that provides two main features:

- **Modular importing** This feature enables the ontology developer to import modules extracted dynamically based on the usage of external terms.

- **Focused concept viewing** This feature lets the ontology developer see a focused view for one or more concepts of interest in an ontology. The terms that are relevant for (i.e. in the module of) the selected concepts are shown in the UI and the rest of the concepts are filtered out.

Both features are implemented using the syntactic locality-based modularization algorithm described in Section 4.2.3. We are planning to implement semantic locality algorithm and compare the differences in extraction time and module sizes. Our design is not hard-wired for a specific modularity algorithm and in principle any modularity algorithm with similar characteristics can be used to provide these services.

The plug-in is still in the very early stage of development. We are investigating the feasibility of some features and might need to scale down the features if we cannot achieve (near) real-time performance for large ontologies. In the following sections, we will describe the current design of the plug-in and explain in more detail how the aforementioned features will work. We will provide a more thorough documentation of the plug-in once the implementation finishes.

6.1 Partial ontology importing

The modular ontology importing feature we are implementing do not require any syntactic and semantic extension to OWL and works seamlessly with current OWL ontologies. The modularity plug-in examines the import statements in an ontology and finds the external terms “used” from the imported ontology. A term is considered to be used if it is referred in any logical axiom in the importing ontology. The plug-in provides customized views for entities (classes, properties, and individuals) that will show only the external entities that are either directly referenced or are in the same module as the directly referenced entity.
The customized entity views are updated automatically as an axiom using
a new external entity is added or all the axioms that uses the external entity
are removed. It is also possible to set the preferences of the plug-in such that
the user will be notified of such changes. The change notification lets the on-
tology developer follow how the imported module evolve based on the axioms
added/removed. This feedback mechanism would let the user immediately see
the non-obvious effects of her changes, e.g. see which other classes get included
in the module because of the directly used class.

The ontology developer is also notified if an external entity is used in a
way violating the safety condition explained in Section 3. As we discussed in
Section 3, non-safe usage of an imported term indicates that the meaning of
that term might be changed in the local ontology which is not desirable most
of the time. But we also recognize that unsafe usage of imports is unavoidable
in certain cases so the tool only indicates unsafety when it occurs but does not
enforce safety automatically. The safety check determines and displays which
axioms need to be changed to guide the user in fixing unsafety.

Another view we are developing is a summary of partially imported terms.
This view shows which external terms from the imported ontology are directly
referenced in the local ontology and which other terms have been indirectly
included in the imports.

6.2 Focused concept viewing

The purpose of focused concept viewing is to facilitate working with subsets of
large ontologies without physically exporting the subset. The ontology developer
selects one or more classes from the ontology and the tool computes the modules
for selected concepts. The views in the hierarchy are adjusted so that only the
terms or axioms that are in the module shown while the rest are filtered out.

With focused concept viewing, one can easily see the parts of ontology rele-
vant for the domain they are interested in. Such focused views can be defined
dynamically and different modules can be assigned to different ontology devel-
opers for maintenance and authoring. Safety analysis can be used for focused
views, too. While in a focused view, if a change in the ontology affects some-
thing outside the focused module we would detect this with the safety check
and warn the user.

7 Acknowledgments

This paper is written with funding from National Cancer Institute (NCI) under
Subcontract No. 27XS075 (S07-075). The opinions and comments expressed in
this paper reflect the views of Clark & Parsia technical team. Our analysis and
final recommendations were highly influenced by the work of Bernardo Cuenca
Grau et al. [16, 15, 17, 10, 11, 14, 12, 13, 19] which provides the formal definition
of modules we used and the key points of the desiderata for modules we adopted.
A Modular ontology development practices

In this section, we will review some ontology development methodologies that help when building modular ontologies. As we mentioned in Section 2, these methods can be used in addition to analytic services we reviewed in this paper.

The simplest methodology for modular ontology development is to split the domain knowledge into some main areas and develop each component in a separate file. This practice is very common and is currently possible by using the owl:imports construct. As we discussed earlier, owl:imports gives a false sense of modularity since tools end up including the imported ontologies as a whole. Earlier in this paper we discussed how modular imports can solve this problem. And even though modular imports can provide this service dynamically without any file separation, a high level partitioning of the ontology contents would help developers who want to reuse parts of that ontology.

Note that modularization algorithms can help ontology authors to partition an existing ontology into pieces and to analyze the existing partitions to see if they are really modular, e.g. if they respect import safety. For example, analysis of SWEET ontologies, a set of 10 ontologies about earth sciences published by NASA JPL, reveal that import safety is violated in several situations, which suggests a partial reorganization might improve the overall structure. With modular imports, ontology authors can also publish parts of their ontology relevant for a sub-domain without partitioning the axioms of the ontology into different files. As an alternative, the ontology author would first identify the terms relevant for the sub-domain and could create an empty ontology that modularly imports those terms. Note that this scenario is not attainable with current algorithms due to the problems we described in Section 5.1, but in theory would be possible to achieve if those issue are resolved.

Standard ontology development methodologies may also help with modular design. For example, the OntoClean methodology [18, 32] has been used successfully in practice to help developers find modeling errors in the ontologies. OntoClean defines several meta-properties for classes (based on the notions of rigidity, identity, unity, and dependence) and restrictions on how classes can be related based on this meta-properties. Applying the OntoClean methodology generally reveals that in many cases multiple inheritance is not semantically justified [32]. For example, making the concept Country both a subclass of LegalAgent and GeographicalRegion merges two very different uses of the concept into one and is not desirable. Rector, starting with the goal of modular ontology development, arrives at a very similar conclusion [27]. Rector even goes further and argues that the domain ontology should split into disjoint homogeneous trees where no domain concept has more than one primitive superclass. The homogeneity requirement is in fact what the OntoClean methodology enforced in the above example.\footnote{Rector gives the following example for homogeneity [27]: Making “vascular structure”} However, the strict single superclass requirement
that Rector proposes is a syntactic restriction as he argues multiple inheritance should always be inferred via definition axioms. Thus, we end up having multiple inheritance, but not through explicit superclass relations, so it is not clear how useful this syntactic requirement is.

A simple technique that was employed as early as in 1990’s in developing Ontolingua ontologies [8] is declaring classes in an ontology public or private. These access modifiers are present in most of the major programming languages and are used to make the intentions of the software author clear regarding which parts of the software are considered to be part of the internal design and which parts are considered to be an external interface and thus appropriate for others to reuse and modify. A similar mechanism is considered to be very useful for ontologies, not only for improving ontology reuse [26], but also for “semantic encapsulation” [3], e.g. when the author deliberately wants to hide the detailed parts of an ontology in order to provide a simpler query interface.

Using access modifiers with OWL classes would not necessarily require a change to the base language. For example, defining a set of annotation properties as in the spirit of Dublin Core (DC) vocabulary[6] would be enough to build best practices regarding these modifiers. An ontology tool would recognize these modifiers and would warn (or even prohibit) if one tries to import and reuse a private term from an external ontology.

---

References


