Ontology Language Extensions to Support Localized Semantics, Modular Reasoning, and Collaborative Ontology Design and Ontology Reuse

Jie Bao
Iowa State University

Vasant Honavar
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Artificial Intelligence and Robotics Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/243

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
Ontology Language Extensions to Support Localized Semantics, Modular Reasoning, and Collaborative Ontology Design and Ontology Reuse

Jie Bao and Vasant Honavar

Artificial Intelligence Research Laboratory
Computer Science Department
Iowa State University
Ames IA USA 50010
{baojie,honavar}@cs.iastate.edu

Abstract.
Modular approaches to design and use of ontologies are essential to the success of the Semantic web enterprise. We describe P-OWL (Package-based OWL) which extends OWL, a widely used ontology language that supports modular design, adaptation, use, and reuse of ontologies. P-OWL localizes the semantics of entities and relationships in OWL to modules called packages. P-OWL and the associated tools will greatly facilitate collaborative ontology construction, use, and reuse.

Keywords: Modular Ontology, Contextual Ontology, Package-extended Ontology, Semantic Web, OWL, P-OWL

1 Introduction
Semantic Web [BL2001] aims to support seamless and flexible access, use of semantically heterogeneous, networked data, knowledge, and services. The success of the Semantic Web enterprise relies on the availability of a large collection of domain or application specific ontologies and mappings between ontologies to allow integration of data [RCH2003; BDS2003] as well as components of complex workflows [PCH2004]. Increasing need for sharing of information and services between autonomous organizations have led to major efforts aimed at the construction of ontologies in many domains e.g., the gene ontology (www.geneontology.org) [A2000] in biology.

By its very nature, ontology construction is a collaborative process which involves direct cooperation among individuals or groups of domain experts or knowledge engineers or indirect cooperation through reuse or adaptation of previously published, autonomously developed, very likely, semantically heterogeneous ontologies. Despite this, relatively little attention has been paid to formalisms and tools for collaborative construction in such settings. This state of affairs in ontology languages and ontology engineering is reminiscent of the early programming languages and first attempts at software engineering when uncontrolled use of global variables, spaghetti
code, absence of well-defined modules leading to unwanted and uncontrolled interactions between code fragments.

Hence, there is an acute need for approaches and tools that facilitate collaborative modular design, adaptation, use, and reuse of ontologies. The lack of such tools is a major barrier to realizing the full potential of the Semantic Web. Against this background, this paper describes P-OWL (Package-based OWL) which extends OWL, a widely used ontology language that supports modular design, adaptation, use, and reuse of ontologies and Ontomill, a collaborative ontology-building tool that includes an ontology editor and a reasoner. The rest of the paper is organized as follows. Section 2 describes some of the requirements of collaborative ontology design tools to motivate the work described in this paper. Section 3 presents the basic definitions and semantics of package-extended ontology; Section 4 describes the reasoning algorithm in package-extended ontology; Sections 5 describes the syntax specifications for package-extended ontology and a possible extension to OWL/RDF, called P-OWL; Section 6 concludes with a brief summary, discussion of related work and some directions for ongoing and future work.

2 Desiderata of Collaborative Ontology Tools

Consider the task of building an ontology for a large state university system. Typically, multiple relatively autonomous groups (faculty, programs, departments, colleges) contribute parts of such an ontology that pertain to their domains of expertise or responsibility. The ontology for the university system should be a semantically coherent integration of the constituent ontologies developed by the individual groups. Hence, there is a need for collaborative ontology construction tools. We enumerate below, some desiderata of such collaborative ontology construction tools.

**Local Terminology:** Terms used in different ontologies e.g., the department name, research topics, and graduate student status, etc. should be given unique identifiers. This is necessary to avoid name conflicts when merging two independently developed ontologies and to avoid unwanted interactions among modules. For example, one individual might define `TurkeyStudy` (the study of the country Turkey) as `AsianStudy` (study of Asia) with `(inRegion = Turkey)`; whereas another individual or group may unknowingly define `Turkey` as subclass of `Asia`. Manual processing of such name conflicts does not scale up with increase in size, number, and complexity of ontologies.

**Localized Semantics:** Collaborative ontology construction requires different groups to adapt or use of ontologies that were independently developed by other groups. However, unrestricted use of entities and relationships from different ontologies can result in serious semantic conflicts, especially when the ontologies in question represent local views of the world from the respective points of view of the ontology producers. For example, university A may define `A:AsianStuy` and `A:EuropeanStudy` as two disjoint concepts in its ontology:

\[ A:\text{AsianStudy} \cap A:\text{EuropeanStudy} = \emptyset \]
whereas University B may define
B: AsianStudy ⊑ A: AsianStudy
B: EuropeanStudy ⊑ A: EuropeanStudy
B: TurkeyStudy ⊑ B: AsianStudy ∩ B: EuropeanStudy
This will lead to obvious semantic conflicts if both ontologies have global semantics.

**Ontology Evolution:** Ontology construction is usually an iterative process. This is especially true in emerging areas of science in which there is little consensus concerning the basic entities and assumed relationships among entities (i.e., ontological commitments). A small change in one part of an ontology may be propagated in an unintended and hence undesirable manner across the entire ontology. For example, two universities A and B initially define ontologies that satisfy the following axioms:
A: AsianStudy ∩ A: EuropeanStudy = ∅
B: AsianStudy ⊑ A: AsianStudy
B: EuropeanStudy ⊑ A: EuropeanStudy
B: TurkeyStudy ⊑ A: TurkeyStudy
But now university B decides TurkeyStudy should be viewed as a kind of EuropeanStudy, by adding a new axiom
B: TurkeyStudy ⊑ A: EuropeanStudy
This will lead to the unintended effect that B: TurkeyStudy to be an empty concept i.e., one with no members or instances.

**Distinction between Organizational and Semantic Hierarchies**
Ontologies are often organized in the form of subsumption (ISA, subclassOf) hierarchies defined over classes and properties. For example, given
HistoryDepartment ⊑ AcademicDepartments
HistoryDepartmentHall , Building
HistoryStudentClubs ⊑ StudentOrganization
Suppose that it is now desired to state that the above three concepts are all about history department. If we were to introduce a new common super-class, say HistoryDepartmentRelated, for the classes that correspond to the three concepts, it will, instead of clarifying the semantics associated with the concepts in question, will introduce logical ambiguity. This is because HistoryDepartmentHall is declared to be an instance of the Building concept, it will (through subsumption), be an instance of HistoryDepartmentRelated. This problem is even worse for properties because of the distinction between datatype property (with range of predefined datatype) and object property (with range of class or instance of a class) in ontology languages such as OWL. It’s usually hard to design a superproperty when both datatype extension and object type extension are possible in future. When we examine ontologies in many application domains, we find that properties are much less organized compared to classes.

A related problem has to do with organizing instances into a hierarchy. For example, if there are a dozen of HistoryStudentClubs instances such as CivilWarClub,
WarOf1812Club, RussianHistoryClub, EnglandHistoryClub and so on, it will be more clear to organize them into hierarchy, such as

AmericanHistoryClubs
   CivilWarClub
   WarOf1812Club
EuropeanHistoryClubs
   RussianHistoryClub
   EnglandHistoryClub

However, this is hard to do in ontology languages such as OWL without modifying the ontology schema (T-box) which is not always possible or safe.

In short, an organizational hierarchy of ontology entities may be different in structure from the semantic hierarchy. Furthermore, there might be a need for an organizational hierarchy even when a semantic hierarchy is missing.

Ontology Reuse: In collaborative design of ontologies, it often makes sense to reuse parts of existing ontologies. However, lack of modularity and localized semantics in ontologies forces an all or nothing choice with regard to reuse of an existing ontology. For example, a university library may want to reuse part of Congress Library Catalog ontology in creating its own ontology. Nevertheless, because ontology languages such as OWL do not support the import and reuse of only a part of an existing ontology. Modular ontologies facilitate more flexible and efficient reuse of existing ontologies.

Knowledge Hiding: In many applications, the provider of an ontology may not wish, because of copyright considerations or privacy or security concerns, to make the entire ontology visible to the outside while willing to expose certain parts of the ontology to certain subsets of users. For example, if an ontology provider reuses licensed commercial ontology such as a part of the CYC ontology, the ontology provider may not be able to reveal that part of the ontology to all users.

Proposed Approach

Current ontology languages, like DAML+OIL and OWL while they offer some degree of modularization by restricting ontology segments into separated XML namespaces, fail to fully support localized semantics, ontology evolution, distinction between semantic and organizational hierarchies over concepts and properties, ontology reuse, and knowledge hiding. In this paper, we argue for package based ontology language extensions to overcome these limitations. A package is an ontology module with clearly defined access interface; mapping between packages is performed by views, which define a set of queries on the referred packages. Semantics are localized by hiding semantic details of a package by defining appropriate interfaces (special views). Packages provide an attractive way to compromise between the need for knowledge sharing and the need for knowledge hiding in collaborative design and use of ontologies. The structured organization of ontology entities (classes, properties, instances) in packages bring to ontology design and reuse,
the same benefits as those provided by packages in software design and reuse in software engineering.

3 Syntax and Semantics of Package-extended Ontologies

Current ontology languages are based on description logics (DL). The syntax and semantics of Package-Extended Ontologies is based on description logic based languages. Description logic is a family of knowledge representation language that can be used to represent the knowledge of an application domain in a structured and precise fashion [BHS2003]. The interested reader is referred to [DCM2003] for details of description logic. In this section, we define the syntax and semantics of package extended ontologies.

**Package**

**Definition 1 (Ontology Entity)** An ontology entity is an axiom $e=[C|P|I]$ where $C$ is a class (concept) definition axiom, $P$ is a property (relation) definition axiom and $I$ is an instance (object) definition axiom.

**Definition 2 (Scope Limitation Modifier, SLM)** scope limitation modifier of an ontology entity $e$ is a Boolean function $V_e(r)$, where $r$ is the identifier of a model that refers $e$. Model(r) could access $e$ if and only if $V_e(r) = \text{True}$.

Possible SLMs include but not limited to, *public*, *protected*, and *private*. They provide a controllable way to define accessing interface of a package. Detailed semantics of the SLMs will be given later.

**Definition 3 (Basic Package):** A basic package is a logic model $P_b = <E,V>$ where $E=\{e_i\}$ is a set of entities and $V=\{v_i\}$ is the set of their SLMs.

**Definition 4 (Compositional Package):** A compositional package is a logic model of $P_c = <E,V,P>$ where $E=\{e_i\}$ and $V=\{v_i\}$ are sets of entities and their SLMs and $P$ is a set of basic or compositional packages. For all $P_i \subseteq P$, we say $P_i$ is $\subseteq_N$ (NestedIn) $P_c$. We define $\subseteq_N$ as a transitive property over package such that $P_1 \subseteq_N P_2, P_2 \subseteq_N P_3 \Rightarrow P_1 \subseteq_N P_3$.

Packages could be recursively nested to form a package hierarchy. One advantage of package hierarchy is that both T-Box and A-Box of a logic model (see below for precise definitions) can be structured in an organizational hierarchy, while their semantics could have different hierarchy or no hierarchy at all.

Given a basic or compositional package $P$ and its entity set $E$ and SLM set $V$, we have definition 5-7:

**Definition 5 (SLM-member)** each $e_i \subseteq E$ is called a $v_i$-member of $P$ and denoted as $e_i \subseteq v_i P$

**Definition 6 (Home Package)** $P$ is called the home package of $e_i$ and denoted as $P = \text{HomePackage}(e_i)$. For compositional package, $P = \text{HomePackage}(P_i)$ for all $P_i \subseteq P$. 


Definition 7 (T-Box and A-Box) The subset of all class definitions and property definitions of $E$ is called the T-Box of $P$, the subset of all instance definitions in $E$ are called the A-Box of $P$.

Definition 8 (Default SLMs) three default SLMs are specified as follows:
- $\text{Public } e\,(r) := \text{True}$
- $\text{Protected } e\,(r) := (r = \text{identifier of HomePackage}(e)) \lor \text{Model}(r) \_N \text{HomePackage}(e)$
- $\text{Private } e\,(r) := (r = \text{identifier of HomePackage}(e))$.

Definition 9 (Signature of Package) the signature of a package $P$ is a triple $<\text{CN}, \text{PN}, \text{IN}>$ where $\text{CN}$, $\text{PN}$, $\text{IN}$ refer to the set of all names of classes, properties, and instances with $P$ as their home package, respectively.

Definition 10 (Entity Scope) The scope $S$ of an ontology entity $e$ in package $P$ is the set of models from which $e$ is visible.
$$S(e) = \{\text{model}(r) | SLM_e(r) = \text{True}\}$$
If $e$ is a public-member of $P$, $S(e)$ is the whole universe; if $e$ is a protected-member of $P$, $S(e)$ is $P$ and all its offspring packages; if $e$ is a private-member of $P$, $S(e)$ is only $P$.

Definition 11 (Default Interface) Shallow Default Interface $l_s$ of a package $P$ is a subset of $P$'s signature such that:
$$\text{EN}_i \_ l_s \iff \text{e}_i \_ P \text{ and } V_i(r) = \text{True}, \text{ for } \forall r$$
where $\text{EN}_i$ and $V_i(r)$ are the name and SLM of entity $e_i$. In another word, shallow default interface is composed by the names of all public entities in that package.

Deep Default Interface, or for short, Default Interface, $l_d$ of a package $P$ is the union of its own shallow default interface $l_s$ and the deep default interface of its home package.
$$l_d(P) = l_s(P) \cup l_d(\text{HomePackage}(P))$$

Note that the definition of deep default interface is a recursive one, which means all the visible entities in its parent path are also in $P$'s own default interface. If a package has no home package, its shallow default interface is also its (deep) default interface.

Theorem 1: Default interface of package $P$ corresponds to the set of all entities that are visible from $P$.
$$l_d(P) = \{\text{name of e} | P, S(e)\}$$

Proof: We have:
$$l_d(P) = l_s(P) \cup l_d(\text{HomePackage}(P)),$$
Suppose the ancestors of $P$ are $P_1, \ldots, P_m$. Then we have
$$l_d(P) = l_s(P) \cup l_s(P_1) \cup \ldots \cup l_s(P_m)$$
For $\forall EN, l_d(P)$, we have
Suppose \( EN \) is the name of \( e \). It could be either 
\[
E \cdot \text{public} \; P
\]
or
\[
E \cdot \text{public} \; P_i, \; i = 1, \ldots, m
\]
Both cases imply
\[
\text{For } \forall r \text{ SLM}_e(r) = \text{public}, \; \lambda(r) = \text{True} \Rightarrow \text{SLM}_e(\text{identifier of } P) = \text{True} \Rightarrow P, \; S(e)
\]
Hence,
\[
\delta \{ \text{name of } P, \; S(e) \}
\]

**Definition 12 (Horizon of a Package)** horizon \( \mathcal{H} \) of a package \( P \) is the set of all ontology entities that could be “seen” from \( P \)
\[
\mathcal{H}(P) = \{ e | P, \; S(e) \}
\]
\( \mathcal{H}(P) \) includes all members of \( P \) and all public and protected members of all its ancestor packages.

**Query, View and Interface**

Packages provide a way to modularize an ontology and to localize knowledge. Now we turn to connecting the modules by specifying mappings between them.

A common way to connect ontology modules is the one-to-one name mapping between modules. This is also supported by ontology languages such as OWL via assertions owl:equivalentClass, owl:equivalentProperty and owl:sameIndividualAs. However, this approach to mapping between ontologies is rather limited in terms of the types of mappings that can be specified. In addition, such mappings are reflexive which is not always desirable. We argue that to maintain the local semantics of a package, query-based or view-based mappings provide a better alternative. We introduce such mappings in what follows.

**Definition 13 (Local Interpretation of Package)** A local interpretation of a package \( P \) is a pair \( I = \langle \mathcal{A}, (.) \rangle \), where the concept space \( \mathcal{A} \) contains a nonempty set of objects and the role space \( (.) \) is a function over \( \mathcal{A} \) such that
\[
\begin{align*}
\text{C}_i \in V & \iff C_i \in \mathcal{A} \setminus \mathcal{A}^l \\
P_i \in V & \iff P_i \in \mathcal{A} \setminus \mathcal{A}^l \\
I_i \in V & \iff I_i \in \mathcal{A} \setminus \mathcal{A}^l
\end{align*}
\]

**Definition 14 (Distributed Interpretation of Packages)** a distributed interpretation of a set of packages \( \{ P_i \}, i=1, \ldots, m \) is a family \( \delta = \{ I_i \} \) where \( I_i = \langle \mathcal{A}, (.) \rangle ^i \) is the local interpretation of \( P_i \). The union of all \( \mathcal{A} \) is the distributed concept space \( \mathcal{A}^{d} \) and \( (.)^{d} = \{ \text{functions over } \mathcal{A}^{d} \} \) is the distributed role space.

**Definition 15 (Query)** Given a set of packages \( \{ P_i \} \), and \( e_1, \ldots, e_m \) are some entity names in \( \{ I_e(P_i) \} \). \( I^e \) is the distributed interpretation of \( \{ P_i \} \). A query over \( \{ P_i \} \) is an expression of one of the following forms:
• **Class Query**: \( C_q(x) := f_c(e_1, \ldots, e_m) \subseteq \Delta^d \)
  where \( f_c \) is a unary (one free variable) logic construction function for classes.

• **Property Query**: \( P_q(x,y) := f_p(e_1, \ldots, e_m) \subseteq \langle \rangle^d \)
  where \( f_p \) is a binary (two free variables) logic construction function for properties.

• **Instance Query**: \( I_q := f_i(e_1, \ldots, e_m) \subseteq \Delta^d \)
  where \( f_i \) is a logic construction function with no variable for instances.

The left hand side of the expression is the **definiendum** of the query and the rhs is the **definien** of the query.

**Definition 16 (View)** a view \( W \) over a set of packages \( \{P_i\} \) is a set of queries over \( \{P_i\} \). \( \{P_i\} \) is called the **domain** of the view.

![Figure 1. A Package with multiple interfaces](image)

![Figure 2. A View can be built upon multiple packages and can be referred to by multiple modules](image)

If we do not limit the expressiveness of query, a view can be as complex as any package. In practice, the expressiveness of queries allowed by definition 15 should be restricted in order to ensure tractability of inference. Possible candidates include conjunctive query [SK2003] or disjunction of conjunctive queries [CGL2001].

**Definition 17 (Interface)** an interface \( F \) over package \( P \) is a view over and only over \( P \).

One module can have multiple interfaces as shown is figure 1, which enables multiple ways to reuse a package. For example, to reuse part of an existing ontology such as Congress Library Catalog, multiple interfaces, such as “History” and “Computer Science” could be defined. The resulting interfaces would allow efficient and flexible reuse of the Congress Library Catalog ontology. Views also offer a reusable mechanism to connect packages if they (the views) are defined over multiple packages. Figure 2 shows two packages \( P_1 \) and \( P_4 \) reuse a view \( V_1 \) over two packages \( P_1 \) and \( P_2 \).

**Definition 18 (Signature of View)** the signature of a view \( W \) is the name set of all query definienda in that view.
Since interface is a special kind of view, signature of interface can be defined in the same fashion.

Note that the default interface $I_d(P)$ of a package $P$ is the signature of a simple interface $F$ of $P$ with only equivalency assertions such as $F.e_i \equiv P.e_i$

where $e_i$ is an ontology entity in $P$.

Package-extended Ontology

View and interface controls how knowledge can in one or more packages can be referred to (exported to) by other packages. To complete the connection among packages, we should also specify how knowledge is imported into a package.

**Definition 19.** (Imported) A package $P_1$ is said being imported into a package $P_2$ if the default interface of $P_1$, $I_d(P_1)$, is used in some entity definition axioms in $P_2$. A view $W$ is said to be imported into package $P_2$ if subset of the signature of $W$ is used in some entity definition axioms in $P_2$. The set of all imported packages and views of a package $P$ is called the domain of $P$.

When a package $P$ or view $W$ is imported into a package, note that only the signature of that $P/W$ is used instead of exposing the entity definition axioms. The referring package only takes care of the set of referred names, while semantics of its domain are maintained intact. When reasoning over the semantics of its domain is needed, a reasoning request should be populated to the referred package/view and locally resolved. Thus, locality of semantics is maintained while allowing global reasoning.

**Definition 20.** (Importing Closure) an importing closure $C$ of a set of packages and views ensures that domains of all views and packages in $C$ are also in $C$.

**Definition 21** (Package-extended Ontology): a package-extended ontology $O = <P, W>$ where $P$ is a set of packages, $W$ is a set of views defined on $P$, $P$ and $W$ constitute an importing closure.

4 Reasoning over Package-extended Ontology

Now we briefly discuss how reasoning over a package-extended ontology.

Reasoning in package-extended ontology can be seen as distributed reasoning among autonomous ontology modules where no global semantics is guaranteed. Therefore, the whole reasoning process has to be built on local reasoning offered by individual modules.

We focus on the subsumption problem – the problem of determining if a class is a subclass of another class. Many other reasoning problems can be reduced to subsumption, for example

1. $C$ and $D$ are equivalent $\iff C$ is subsumed by $D$ and $D$ is subsumed by $C$. 
2. C and D are disjoint ∴ C ∩ D is subsumed by ⊥ (bottom concept).
3. a is a member of C = {a} is subsumed by C

First we give the definition of subsumption reasoning in package-extended ontology:

Definition 22 (Interpretation of Package-extended Ontology) interpretation $\equiv_{\langle P, W \rangle}$ of a package-extended ontology $O = \langle P, W \rangle$ is the distributed interpretation of $\{P, W\}$ if $W$ is treated as packages.

Definition 23 (based on [SK2003]) (subsumption reasoning request) over package-extended ontology $O$ with respect to some interpretation $I$ involves checking whether a class $C$ is subsumed by another class $D$ with respect to $I$, denoted as $C \sqsubseteq D$ iff $I \vdash C \sqsubseteq D$. We say $C \sqsubseteq D$ if $C \sqsubseteq D'$ for all possible $I$.

We describe an extension of the Tableau algorithm in description logic [BCM2003, p78] for subsumption reasoning over a package extended ontology. We restrict our discussion to the language of $ALCN$. The general idea of standard Tableau algorithm is to reduce the subsumption problem to (un)satisfiability problem and try to construct a possible interpretation for given terminology. The reduction is easy to understand since $C \sqsubseteq D$ iff $C \sqsubseteq D'$ is unsatisfiable. Transform $C \sqsubseteq D$ into negation normal form (NNF), i.e. negation occurs only in front of concept names. Denote the transformed expression as $C_0$, the algorithm starts with an ABox $A_0 = \{C_0 \{x_0\}\}$, and apply consistency-preserving transformation rules [BCM2003, p81] to the ABox as far as possible. If one possible ABox is found, $C_0$ is satisfiable and the subsumption is not true. If no possible ABox could be found, the subsumption is true.

The algorithm for distributed subsumption reasoning is as follows:

**SubsumptionAnswer** (C, D, O)
Input: Concept C and D, Ontology $O = \langle P, W \rangle$
Return: True or False
1. Construct an ABox $A = \{C \cap D (x)\}$, Transform A into NNF.
2. FOR all package/views $P$ being referred in $A$
3. RETURN Satisfiable ($\{A\}$, $P$);
4. END FOR

**Satisfiable** (S, P)
Input: Initial ABox set $S$, package/view $P$
Return: True or False
1. FOR all ABoxes $A_i$ in $S$
2. Transform concepts in $A_i$ into NNF wrt visible entities from $P$;
3. Do ABox transformation as that in standard tableau algorithm, result in an augmented set of ABoxes $S_o$, $S' = S \cup S_o$.
4. IF $\exists A \in S_i$ is complete and consistent
5. RETURN True;
6. ELSE
FOR all imported packages/views $P$

IF Satisfiable ($S' , P$) = True

RETURN True;

END IF

END FOR

END IF

END FOR

RETURN False;

An ABox is called complete if none of the transformation rules applies to it. An ABox is called consistent if no logic clash is found.

The basic idea of Satisfiable algorithm is that a package or view could answer a Satisfiable request if a possible interpretation is found locally; otherwise it will consult the packages and views in its domain. Although no global semantics is available, an interpretation of the “global” model is incrementally constructed by the queries among packages and views.

Suppose the domain of each module (package or view) is finite and expanded importing path for every package has finite length and no cyclic importing is allowed, the final call times of Satisfiable is PSPACE-complete. It is easy to prove from the properties of the Tableau Algorithm that the SubsumptionAnswer algorithm is sound, terminable, complete and decidable, given all modules are limited with ALCN-concept description. Since we know satisfiability of ALNC-concept description is PSPACE-complete in each of the package, the SubsumptionAnswer is also PSPACE-complete for this case.

5 Specifications of P-OWL Language.

In this section, we show the basic formalism of package-extended ontology can be incorporated into ontology languages such as OWL. Also, to keep backward compatibility to legacy systems, we want the extended ontology language is syntactically as compatible as possible with existing ontology languages. Hence, instead of introducing new syntax, a large part of this specification is given in OWL, RDF and RDFS to extend OWL. The part of the specification that cannot be given in OWL and RDF is specified using rules.

To allow tradeoff between expressiveness and complexity, the proposed solution is offered in two versions. The Lite version enables basic package, view and interface functionalities, thereby providing support for modularity and information hiding capacity; The Full version supports composition of packages.

P-OWL Lite specifications

Spec 1. Package is defined inside a XML namespace; one namespace can hold multiple packages

Spec 2. Package is a special OWL class
Spec 3. There is one and only one global package \( P^0 \)

- \( P^0 \in \text{Package} \)

Spec 4. Each term belongs to a unique package. If not explicitly stated,
   a) owl:Thing and owl:Nothing has assumed package \( P^0 \)
   b) For class, the homepage package of superclass is assumed as home package; if no superclass, \( P^0 \) is assumed
   c) For property, the homepage package of superproperty is assumed as home package; if no superproperty, \( P^0 \) is assumed
   d) For instance, the homage package of the class type is assumed as home package.

- inPackage \( \sqsubseteq \text{rdf:Property} \)

- range(inPackage) = Package

- \( x = \{ \text{Owl:Thing} | \text{owl:Nothing} \} \rightarrow \text{inPackage}(x, P^0) \)

- SubClassOf(x,y) \cap \text{inPackage}(x, \)"\) \cap \text{inPackage}(y,z) \rightarrow \text{inPackage}(x,z)

- SubPropertyOf(x,y) \cap \text{inPackage}(x, \)"\) \cap \text{inPackage}(y,z) \rightarrow \text{inPackage}(x,z)

- \( x \in y \cap \text{inPackage}(x, \)"\) \cap \text{inPackage}(y,z) \rightarrow \text{inPackage}(x,z)

Spec 5. No entities in one package can have identical local names; No package names in one namespace could be identical.

To make it compatible to OWL language, an entity is given a unique storage name with its package name as prefix. The translating from package/local name to storage name should be supported by the ontology editor and reasoner. For example, an entity named “OWL” defined in package “Language” could be stored as “Language/owl” and an entity named “OWL” defined in package “Animal” could be stored as “Animal/owl” in the same ontology.

Spec 6. Each term has a SLM. Possible SLMs include
   a) Public: terms is visible to the whole universe
   b) Private: term is visible inside this package only.
   Default modifier is public

- \( \text{InPackagePublic} \sqsubseteq \text{inPackage} \)

- \( \text{InPackagePrivate} \sqsubseteq \text{inPackage} \)

- \( \text{InPackage}(x,p) \cap \text{InPackagePrivate}(x,p) \rightarrow \text{InPackagePublic}(x,p) \)

Spec 7. Default interface of a package is the public entities in that package

- \( \text{DefaultInterface}(p) := \{ "x | \text{InPackagePublic}(x,p)\} \)
Spec 8. Query is an equivalency statement in the form

\[ x \equiv f(p_1::y_1, \ldots, p_m::y_m) \]

where \( x \) is a new defined entity, \( f \) is a logic construction function legal in OWL, \( p_i \) is referred package and \( y_i \) is referred entity in \( p_i \). The actual equivalency axiom can be one of the following:

- \( \text{owl:equivalentClasses} \)
- \( \text{owl:equivalentProperties} \)
- \( \text{owl:SameIndividual} \)

Only visible entities in a package could be directly queried.

\[ y_i \in \text{DefaultInterface}(p_i) \]

Spec 9. View is a special package with all its members public. Query is the legal axiom in a view. All referred packages should be explicitly imported.

\[ \text{View} \sqsubseteq \text{Package} \]
\[ \text{importPackage} \sqsubseteq \text{owl:ObjectProperty} \]
\[ \text{Range(importPackage)} \sqsubseteq \text{Package} \]
\[ \text{Domain(importPackage)} \sqsubseteq \text{View} \]
\[ \text{inPackage}(x,v) \cap \text{View}(v) \rightarrow \text{inPackagePublic}(x,v) \]
\[ \text{inPackage}(x,v) \cap \text{View}(v) \rightarrow \text{RDF statement with } x \text{ as subject is an equivalency statement} \]
\[ \text{inPackage}(x,v) \cap p \text{ is referred in statement about } x \cap \text{View}(v) \cap \text{Package}(p) \cap \text{importPackage}(v, p) \rightarrow \text{TRUE} \]

Note that View could also be referred as a normal package.

Spec 10. Interface is a special view such that all its queries are submitted to one and only one package.

\[ \text{Interface} \sqsubseteq \text{View} \cap \text{importPackage}=1 \]

Spec 11. Package except view can only refer to other packages by views, especially by interfaces. However, the default interface of a package could be used without explicit definition. When refer an entity via a view, both the entity’s local name and the view’s name should be given.

\[ \text{importView} \sqsubseteq \text{owl:ObjectProperty} \]
\[ \text{Range(importView)} \sqsubseteq \text{View} \]
\[ \text{Domain(importView)} \sqsubseteq \text{Package} \cap \text{View} \]
\[ \text{inPackage}(x,p) \cap v \text{ is referred in statement about } x \cap \text{View}(v) \cap \text{Package}(p) \cap \text{importView}(p, v) \rightarrow \text{TRUE} \]
Spec 12. No cyclic reference on entities is allowed. However, cyclic importing of packages and views is allowed if no cyclic entity reference is included.

Since the entities are stored in storage name, the OWL validator can be used to check for cyclic definition.

P-OWL Full specifications

In addition to the specifications stated above for the Lite version, the Full version includes following specifications to support package composition.

Spec 13. Any package that is not a view can be a subpackage of another package; subPackageOf is a transitive property

- SubPackageOf \(\subseteq\) InPackagePublic
- Range(subPackageOf) \(\subseteq\) Package \(\cap\) View
- Domain(subPackageOf) \(\subseteq\) Package
- SubPackageOf(p,q) \(\cap\) SubPackageOf(q,r) \(\rightarrow\) SubPackageOf(p,r)

A convention of naming package is the hierarchical prefix-naming like that in java.

Spec 14. An extra scope modifier “protected” is introduced when state entities in a package. Entity with “protected” is invisible to the outside except to subpackages.

- InPackageProtected \(\subseteq\) inPackage
- InPackage(x,p) \(\sqcap\) InPackagePrivate(x,p)
  \(\cap\) InPackageProtected(x,p) \(\rightarrow\) InPackagePublic (x,p)
- InPackageProtected(x,q) \(\cap\) SubPackageOf(p,q) \(\rightarrow\) InPackageProtected(x,p)

Spec 15. Entities inside superpackage with modifiers “public” or “protected” could be referred by subpackages, but entities directly defined in subpackages couldn’t be referred from superpackage.

- InPackagePublic(x,q)) \(\cap\) SubPackageOf(p,q) \(\rightarrow\) InPackagePublic (x,q)

“Protected” case has already been given in the last specification. The latter part of this specification is to say

\[\text{InPackage } (x,p)) \cap \text{SubPackageOf}(p,q) \cap \text{InPackage } (x,q) \rightarrow \text{InPackage } (x,q)\]

It is always trivially true.

Spec 16. InnerInterface of a package is the public and protected entities in that package
• InnerInterface(p)

\{ "x| InPackagePublic (x,p) \| InPackageProtected(x,p) \}

Spec 17. Entities in the default/inner interface of a package is also in the
default/inner interface of all the subpackages of its home package.

• x ∈ DefaultInterface(q) \cap SubPackageOf(p,q) \rightarrow x ∈ DefaultInterface(p)
• x ∈ InnerInterface(q) \cap SubPackageOf(p,q) \rightarrow x ∈ InnerInterface(p)

are always true by Spec 7, Spec 14, and Spec 15.

Spec 18. Reference to entities in a package inner interface do not need to be via a
view. That means that a subpackage can refer to public and protected entities in
superpackages without view or interface. However, an entity's home package
needs to be specified.

Part of the specifications is given in rules and cannot be formalized in RDF or OWL.
This part of specifications should be abided by the P-OWL compatible ontology
editors and reasoners. However, a logically sound representation of those rules should
be implemented by some rule language, such as RuleML (http://www.ruleml.org) or

The RDFS-compatible specifications for P-OWL is given in RDF schema and can be
accessed at http://boole.cs.iastate.edu/indus/owlmodule#. Figure 3 gives a P-OWL
based ontology example. An ontology editor with the support of package has also
been developed.

6 Summary and Discussion

Summary

Main contributions of this paper include:
• Package-extended ontologies to support localized semantics, controllable
  knowledge hiding as well as knowledge sharing, modular reasoning, and
  collaborative ontology construction
• A mechanism for view-based information integration over modular ontologies with
  localized semantics.
• A distributed reasoning algorithm over package-extended ontology.
• Specification of P-OWL, a OWL/RDF compatible language for package-extended
  ontologies to support package-extended ontologies.
Related Work

Distributed Logics

A number of distributed logics systems have been studied during recent years, such as Local Model Semantics [GG2001] and Distributed First Order Logic (DFOL) [GS1998] in which both the local semantics and the compatibility relations among local models is emphasized. Distributed Frame Hierarchy, which is a collection of state functions of individual sub-hierarchies connected by combination operators has been proposed in [S2002-1] and [S2002-2].

Partition-based Logics provide approaches to decompose knowledge base into smaller components. [AM2000] studied how to automatically decompose prepositional and first-order logic into partitions and a reasoning algorithm with partitions using message passing. [R2003] proposed an approach to decompose ontology into independent disjoint skeleton taxonomies. These approaches start from a global centralized logic system and try to divide it into components. In contrast, our focus is on collaborative design of large ontologies using independently developed ontologies. Inspired by DFOL, [BS2002] extend the description logic into a distributed scenario.

A Distributed Description Logic system is a set of distributed TBoxes and ABoxes connected by "bridge rules". One important feature of bridge rules is that it is unidirectional so that no "back-flow" is allowed when connecting modules. One limitation is the bridge rules are defined binary correspondences thus restricted it from more expressive articulation between ontology entities, compared with view-based approaches. When the number of involved modules become large, the explicit declaration of such bridge rules becomes tedious.

Modular Ontologies

Two directions have been developed based on DFOL and DDL. One is Modular Ontology, [SK2003] and [SK2003-2] gives a fundamental definition of modularization of ontology and exploitation of modularity in reasoning. It also defines an architecture that supports local reasoning by compiling implied subsumption relations. In this context, the problem of maintaining the semantic integrity of the ontology when it undergoes local changes was studied. A "view-based" approach in integrating ontology in that all Eternal Concept Definitions are given in a set of queries. However, A-Box is missing in their query definition, and the mapping between modules is unidirectional thus local semantics cannot be preserved.

Contextual Ontology

Contextual logic which is based on distributed description logics emphasizes localized semantics in ontologies. Contextual ontology keeps contents local and maps the content to other ontologies via explicit bridge rules. [BDS2002] proposed C15ML, which includes a hierarchy-based ontology description and a context mapping syntax. [BGH2003] combined C15ML and OWL into Context OWL (C-OWL), which is the
syntax for bridge rules over OWL ontology. Several specific bridge rules such as “equivalent”, “onto”, “into”, “compatible” and “incompatible” were defined. Application of C-OWL in medical ontologies was reported in [SHS2004]. The improvement of P-OWL over C-OWL is the introduction of scope limitation modifier and query-base view. Bridge rules could be seen as special cases of query and SLM offers a controllable way to keep content local by definition.

**Ontology Integration Systems**

*Ontolingua* [G1992][FFR1997] was one of the earliest systems that supported manipulating modular ontology. Modular reusable ontologies could be assembled and extended by *inclusion, polymorphic refinement, and restriction*. The scope of symbols in ontologies could also be restricted as being *public or private*. Ontolingua is based on an extended KIF language with frame ontology. Ontolingua does not support localized semantics and mapping between modules is significantly simpler (and hence less expressive) than that supported by our approach.

*ONION* [MWK2000] adopted a graph-based model to represent ontologies. Different models are connected by semantic implication bridges such as $O_1.A \Rightarrow O_2.B$. A simple set of algebra is defined to enable interoperation between ontologies using the articulation ontology. Major limitation of ONION is that the mapping is on the name-to-name basis. ONION basically organizes ontologies into a hierarchy with a top articulation ontology and an implied global semantics. There is no support for localized semantics.

View based integration has been well studied in database research community [H2001]. However, view-based integration of ontology is relatively under explored. This is partly because of the lack of adequately expressive yet efficient query languages for ontologies. [HT2002] shows that conjunctive query language can provide reasonably expressive query language for DAML+OIL or any other description logic based ontology language. A number of proposals for query languages over ontology language such as RQL and RDQL have begun to appear in the literature. *KAON* includes a view language for RDF based on RQL [VOS2003-1] [VOS2003-2]. However, these languages return only extensional results (a set of instances) rather than intentional query needed for constructing view among ontology modules.

[CGL2001] proposed a view-based query answering mechanism for ontology integration. An Ontology Integration System (*OIS*) is a triple $<G, S, M_{G,S}>$ where $G$ is the global ontology, $S$ is the set of local ontologies and $M_{G,S}$ is the mapping between $G$ and the local ontologies in $S$. The mapping can be global-to-local or local-to-global. However, the assumption of the existence of a global ontology is too strong for real world Semantic Web applications where no global semantics can be guaranteed across independent, semantically heterogeneous and autonomous information sources.

**Future Work**
Some directions for ongoing and future work include more careful investigation of the reasoning algorithm and its extension to more powerful DL language such as SHIQ (the DL language used by OWL); the study the basic operations needed in reasoning with package and view, such as the construction of default interface and horizon for a package, checking if an entity is in the default interface of a package (visible to the outside); Efficient representation of mapping between packages; Implementation of tools to support P-OWL, such as ontology editor and reasoner.

Acknowledgement
This research is supported in part by grants from the National Science Foundation (0219699) and the National Institutes of Health (GM 066387) to Vasant Honavar The authors are grateful and members of the Artificial Intelligence Research Laboratory and Dr. Gary Leavens of the Department of Computer Science at Iowa State University for helpful discussions concerning some of the ideas presented in this paper.

References


