Alignment-based modules for encapsulating ontologies*

Jérôme Euzenat[1], Antoine Zimmermann[1], and Fred Freitas[2]

[1] INRIA Rhône-Alpes, Montbonnot, France
[2] UFPE, Recife, Brazil

Abstract. Ontology engineering on the web requires a well-defined ontology module system that allows sharing knowledge. This involves declaring modules that expose their content through an interface which hides the way concepts are modeled. We provide a straightforward syntax for such modules which is mainly based on ontology alignments. We show how to adapt a generic semantics of alignments so that it accounts for the hiding of non-exported elements, but honor the semantics of the encapsulated ontologies. The generality of this framework allows modules to be reused within different contexts built upon various logical formalisms.

1 Introduction

Reusing knowledge was at the origin of introducing ontologies in the Knowledge sharing project. In addition, sharing is one of the main feature of the web. However, in the current state of the semantic web, reuse is difficult to achieve with available tools. In particular, the recommended ontology language for the semantic web, OWL, features an `owl:import` primitive that fails to provide much control on what is imported. From an ontology engineering point of view, this is a major drawback of the definition of ontologies in the semantic web.

Our goal is to design a module system for ontologies that brings to ontology engineers typical properties of software engineering modules. Moreover, we want it to be general enough to be adapted to any logical formalism. Software engineering modules are based on the separation between the interface and the implementation of a module. The interface describes the entities of the module that can be used outside of the module while the implementation is not available outside but through this interface. This helps controlling what is provided by a module. The properties typically expected from a module system are:

Encapsulation enforces module independence by "hiding" implementation details to importing modules. Module specifications are described as an interface which is what can be called by importing modules. So module implementation can evolve without affecting the importing module specification and modules are replaceable by others offering the same interface. In terms of ontologies, the implementation corresponds to the set of axioms used to define it.

* Jérôme Euzenat and Antoine Zimmermann have been partly supported by the European NeOn integrated project (IST-2004-507482).

Separate development. Since what imported modules provide is well-defined, software developers can rely on this interface and develop their own part. This would be useful as well for ontology development in which one can concentrate on the development of part of the model while the part it is relying on are still underspecified.

Separate compilation decreases development time by avoiding recompiling huge applications at every change. This has has been ackled by [15].

Reusability. Because module specifications are descriptions of a coherent and explicit set of primitives that a module is meant to provide, a module can be reused in another similar context.

Separate execution allows to be more efficient by executing local code first. In ontology engineering, this is especially useful for inference system [9].

This paper describes a framework for making modules that can offer most of the features of software engineering modules. Other features like modularization of existing ontologies or optimizing particular properties (*e.g.*, inference needs) are not considered here. As far as designing modules is concerned two approaches can be considered:

Composing ($M \Leftarrow M'_1 \otimes \ldots M'_n$) is the favorite approach in software engineering where a software is built by assembling modules;

Partitioning ($O \Rightarrow O'_1 \oplus \ldots O'_n$) is the back-up strategy when one starts with some monolithic software (or ontology) and attempts at transforming it into a modular software. Depending on the reasons to do this (allowing separate inference or generating modules for reuse), the results may be different.

Some other approaches aim at identifying the subparts of ontologies required for interpreting some particular piece of knowledge [4]. These techniques do satisfy a momentary need and are not really relevant to ontology engineering preoccupation in which modules are defined for being reused.

We adopt here an ontology engineering approach to modules: we do concentrate on how to specify reusable modules rather than how to partition legacy ontologies. Of course, for this to work correctly, the perimeter of each module must be precisely defined and interfaces between modules be precisely designed. This requires to be able to express what is visible or hidden, thanks to the definition of interfaces. Moreover, in order to reuse modules designed independently, the module infrastructure has to provide means of relating them in a consistent way. This is achieved by specifying—or referring to existing—ontology alignments in the modules. This way, the module framework can take advantage of ontology alignment technologies, in particular alignment composition. These modules can replace ontologies wherever they are used. For that purpose, we must define their semantics, *i.e.*, what are the consequences of an ontology module. We propose a syntax for specifying explicit modules that can be reused and provide the hiding properties that have been considered and that allows relating modules with alignments. We give a thorough overview of existing module semantics and analyzes their appropriateness with regard to the desired features. We show that our module framework can be adapted to most of these logical formalisms.

We will first motivate this work with a concrete example in Sect. 2. We will then provide a syntax (Sect. 3) and describe a generic semantics (Sect. 4) that can be adapted

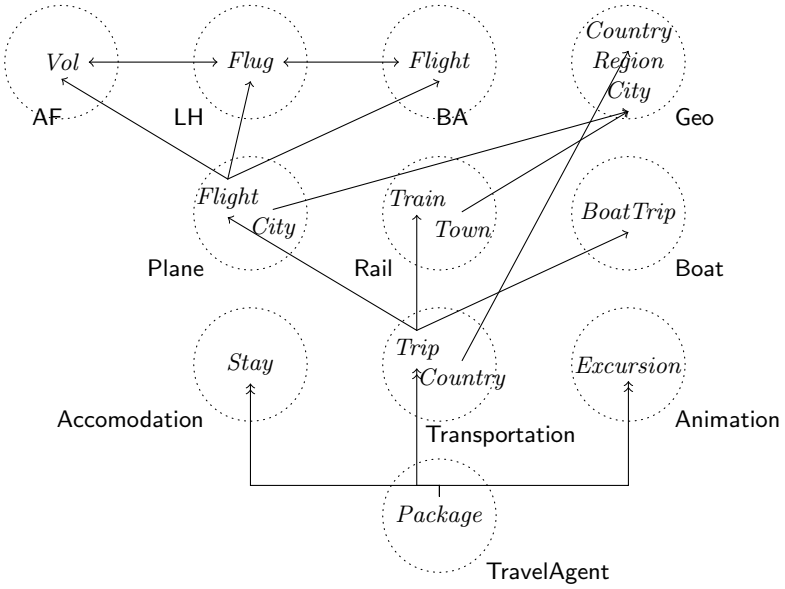

Fig. 1. Dependencies between classes and properties defined in the travel agent example. Circles represent modules that contains *entities*, simple arrows represent correspondences between entities and double arrows mean that target *entities* are components of the source *entity* (and also denote the import of a module by another one).

to most of the existing module languages. Last, in Sect. 5, we discuss the advantage of using such or such formal semantics with regard to the desiderata we provided here, and finally compare this work to other work on the same topic, that have been recently published.

2 Motivating example

Imagine an integrating application like a travel agent that sells packages including travel tickets and hotel stays, among other products and services. It is necessary that this agent be able to use information provided by other sources. For that purposes, it will take advantage of other modular ontologies. For instance, this TravelAgent module will provide packages including trip, accommodation and excursion and will rely on other more qualified sources to define these facilities that it sells. These imported modules will provide the necessary information about Transportation, Accommodation and Animation. This is displayed in Fig. 1.

Defining the module's imports: A module like TravelAgent is an aggregating module that generates new services by aggregating previously available information. Transportation is a module that provides a homogeneous interface to other heterogeneous ontologies. For instance, this ontology could rely on several ontologies provided by specific

providers: Plane, Rail, Boat, for transportation means ontologies and Geo for geographic information. So our module language should allow to express that:

– *Module* Transportation *uses Modules* Plane, Rail, Boat *and* Geo.

In fact, Transportation will only import specific information from the other modules: $destinations$ and $origin$ properties of the $Flight$ class but not $planeNumber$; $BoatTrip$ from the Boat module, but not $Cruise$. For that matter, the language has to express:

– *Module* Transportation *imports Class* $Flight$ *and Properties* $originCity$, $destinationCity$, $company$, $price$, tax *and* $date$ *from Module* Plane.
– *Module* Transportation *imports Class* $Train$ *and Properties* $startStation$, $endStation$, $fees$ *and* $date$ *from Module* Rail.
– *Module* Transportation *imports Class* $BoatTrip$ *and Properties* $boardingPort$, $unboardingPort$, $price$ *and* $date$ *from Module* Boat.
– *Module* Transportation *imports Classes* $City$ *and* $Country$ *and Property* $partOf$ *from Module* Geo.

Declaring the module's content: The goal of the Transportation module is to provide the $Trip$ class which will describe trips between two cities. It will also provide some refined classifications of them such as $DomesticTrip$ and $InternationalTrip$:

– *Module* Transportation *defines Class* $Trip$ *with Properties* $from$ *which must be a* $City$, to *which must be a* $City$, $price$ *which must be an Integer,* $date$ *which must be a Date and* $means$ *which must be a String.*
– *Module* Transportation *defines Class* $DomesticTrip$ *as a kind of* $Trip$.
– *Module* Transportation *defines Class* $InternationalTrip$ *as a kind of* $Trip$.
– *Module* Transportation *exports Classes* $Trip$, $DomesticTrip$, $InternationalTrip$ *and* $Geo{:}City$ *and Properties* $from$, to, $price$, $date$ *and* $means$.

Additionally, the Transportation module will provide more information about its core business, that is, its ontology definition. It will define what is the information that it describes: *i.e.*, $Trip$ and its subclasses:

– *A* $DomesticTrip$ *is a* $Trip$ *which starts* $from$ *the same* $country$ *as it is going* to.
– *An* $InternationalTrip$ *is a* $Trip$ *which is Not a* $DomesticTrip$.

Aligning different modules: the Transportation ontology will have to provide a way to map the export interface of the various ontologies into entities used in its body or in its export interface. This is expressed by correspondences like the following:

– *Class* $Flight$ *in Module* Plane *is a kind of* $Trip$ *in Module* Transportation *such that Property* $means$ *is equal to "plane".*
– *The sum of Properties* $price$ *and* tax *of Module* Plane *corresponds to Property* $price$ *of Module* Transportation.
– *Property* $originCity$ *of Module* Plane *corresponds to Property* $from$ *of Module* Transportation.
– *Property* $destinationCity$ *of Module* Plane *corresponds to Property* to *of Module* Transportation.
– *Class* $Train$ *in Module* Rail *is a kind of* $Trip$ *in Module* Transportation *such that Property* $means$ *is equal to "train".*

– *Class* $BoatTrip$ *in Module* Boat *is a kind of* $Trip$ *in Module* Transportation *such that Property* $means$ *is equal to "boat".*
– *Property* $country$ *of Module* Transportation *corresponds to Property* $partOf$ *of Module* Geo *with a range restricted to Class* $Country$ *of Module* Geo.

Of course, the Plane module itself may be the integration of the ontologies of various providers (LH, AF, BA). No one would require that ontologies modeling these providers are semantically compatible: this is why such correspondences are needed. The case of the Plane module is interesting since it can take advantage of existing available alignments (see Fig. 1). Such alignments could exist in an alignment repository, and the module framework should allow to refer to them. Note that the correspondence involving Property $country$ is not expressible in OWL so far, because range restrictions only apply to classes, not properties.

Summarizing our needs: In summary, in this example, we would like to use some external modules in such a way that:

– What is exported by each module is precisely defined:Transportation exports $Trip$;
– The importing module can restrict the imported entities that are used: TravelAgent can import $Stay$ but not $Meal$ from Accommodation for instance; and
– The imported entities can be rewritten by matching the imported entities to internal forms: a $Flight$ is matched to a $Trip$.

But by doing so we would like that some semantic consequences apply to modules. In particular we want that:

– What is true of a $Flight$ in BA is also true of a corresponding $Trip$ in Transportation;
– Whatever consequence of BA that does involve entities not imported in Transportation is not necessarily a consequence of Transportation.

These intuitions will be made more precise later.

This organisation provides independence between the different modules that participate in the module system. The only contract between two modules is that they provide definition for the elements in their export interface.

In particular, modules do not provide any guarantee on the "implementation" of these modules, *i.e.*, the axioms that govern the concepts defined inside the module. As a result, the implementation of a module can be improved without altering the export interface and thus without preventing the application to work. This is *encapsulation*.

On the contrary, if the export interface is modified in such a way that the application will not be able to work, this can be checked without looking at the axioms. This ensures the *separate development* property for these modules. For instance, the Flight modules may integrate new company ontologies without changing its export interface.

This also allows to *reuse* modules more easily: if a new better module providing Rail information is available, it can, as soon as it provides the same export interface, replace the old one without breaking the whole application. The Geo module has typically been reused from an external source.

These properties require a particular semantics for these modules that differ from the classical semantics that would result from the transitive closure of imports in OWL

Feature	Description	Syntactical representation
id	Module identifier	URI reference
IMPORT INTERFACE		
uses	List of modules	URI references
imported-entities	List of ontology entities from imported modules	URI references or keyword `ALL`
DEFINITIONS		
alignments	List of alignments	URI references or locally defined correspondences
content	Entity definitions and axioms	either an external ontology identified by a URL or axioms that can use imported classes and properties
EXPORT INTERFACE		
exported-entities	List of entities from either the ontology defined in `content` or from the imported entities in `imports`	given by URI references or by keywords `ALL` or `ALL*`
comments	Textual description of what the module is made for	a character string

Table 1. Module features, their meaning and values (the `ALL` keywords are given here for completeness purpose and not used in this paper).

(that is the union of all axioms of the transitive closure of imports). In particular, the consequences of a module can only be formulas made of the concepts and properties in the interface of modules and as soon as a module does not export the entities that it imports, they are not visible anymore from importing ontologies.

We provide a concrete syntax for doing this and we define a generic semantics that encompasses several specific formalisms. We discuss to what extent these formalisms can satisfy the requirements.

3 Syntax

This section introduces the various features that can be put in a module definition and proposes a general module definition for ontology language. We will mostly rely on Description Logics (like OWL) as an ontology language. However, the semantics can accommodate other languages as we show in Sect. 4.

3.1 Module features

A module contains an ontology definition which can use entities of imported modules (*i.e.*, classes and properties). So the module syntax proposed here imposes that imported modules be clearly specified, and only entities from these modules are used, in addition to locally defined ones. In order to correlate potentially heterogeneous imported modules, they are related thanks to ontology alignments, that may be defined locally into the

module definition or referred to with a URI reference. Finally, in order to ensure encapsulation, the module definition explicitly states which entities are exported, *i.e.*, which ones are allowed to be used by other importing modules. Additionally, there should be a textual description of what the exported concepts represent, which should explain what is guaranteed to stay true throughout the evolution of the module. This allows specifying the behavior of the module while not displaying the internal implementation. In particular, only the terms given in the export interface need to be showed to external users. These elements are presented in Table 1.

Of course, the module content (*i.e.*, alignments and ontology) can only refer to imported ontology entities that are part of the import interface. In turn, the import interface can only refer to entities which are part of the export interface of the imported modules.

Alignments and ontologies can be referred to by a URI reference, which means that it is possible to reuse published alignments and ontologies independently of a module definition. In this case an ontology is interpreted without its `owl:import` features.

The graph of imported modules, *i.e.*, the *uses* relation, must be acyclic.

We have defined an RDF/XML syntax that follows the same structure. Instead of using this equation based syntax, it reuses the RDF/XML syntax of OWL and that of the Alignment format [5].[3]

3.2 Abstract syntax

In order to define the semantics of the module system we consider an abstract syntax for the ontology modules that is easier to manipulate.

Definition 1 (Ontology module). *An* ontology module $\mathfrak{M} = \langle \mathrm{id}, M, I, A, O, E \rangle$ *is a sextuple such that:*

- id *is a URI identifying the module;*
- $M = (M_i)_{i \in J}$ *is the set of imported modules defined over a set of indices* J*;*
- $I = (I_i)_{i \in J}$ *is the import interfaces for* M*;*
- O *is the content ontology, defining local terms and local axioms that may use imported terms;*
- $A = (A_{ij})_{i,j \in J}$ *is a set of alignments interconnecting ontologies from* M *or* O*, and*
- E *is the export interface of* $\mathfrak{M}$*.*

A base module encapsulating an ontology will typically be expressed as $\langle \mathrm{id}, \emptyset, \emptyset, \emptyset, O, E \rangle$.

We must now introduce the other elements. Ontologies can be seen as OWL ontologies but, as we will see, the semantics does not need to rely on this assumption. An alignment is a restriction of the definition of alignment found in [5, 6], in which the confidence value is always maximal.

Alignments are based on correspondences which relates entities from two ontologies:

[3] An example of the concrete XML syntax can be found at `http://www.inrialpes.fr/exmo/people/zimmer/module.xml`.

Definition 2 (Ontology alignment). *An ontology alignment between ontologies O_1 and O_2 is a set of triples $\langle e_1, e_2, r\rangle$ such that:*

- *$e_1 \in O_1$ and $e_2 \in O_2$ are ontology elements (e.g., class, properties, individuals) from the two ontologies to align;*
- *r is an alignment relation (taken from a given set $\mathfrak{R}$) that is asserted to hold between e_1 and e_2 (e.g., subsumption, equivalence, disjunction, etc.).*

These triples are called correspondences.

It is clear that any module described with the concrete syntax can be translated in a module description in the abstract syntax.

4 The semantics of modules

The semantics describes how to interpret modules and what are the semantic consequences of a module. In this section, we do not describe a specific semantics, but rather we give generic notions that are common to most of module logical formalisms.

Whichever formalism is used, there is a need to differentiate between local interpretation of a module (*i.e.*, interpretation of the ontological content) and global interpretation which takes into account imported modules and alignments. This section first present the local semantics with very general definition (§4.1). Then, a generic semantics of alignments is proposed (§4.2). Finally, we describe the notions introduced by the global semantics (§4.3).

4.1 Local semantics

Since many ontology languages can be used to specify the local content of a module, we will give a very general definition of interpretation and models of an ontology. However, as in our examples, the ontology language will typically be a Description Logics (*e.g.*, OWL-DL), as it is the case in most of existing modular ontology languages (in fact, all but $\mathcal{E}$-connection [11])

All the modularization formalisms have in common that a local ontology O is characterized by its signature[4] S and a set of axioms built over this signature.

To each ontology language is associated a notion of interpretation $\mathcal{I}$ which is a mapping from the elements of a signature to elements of a domain of interpretation Δ, and a notion of satisfaction $\models_l$ which relates interpretations to the axioms they satisfy[5]. A model of an ontology O is an interpretation $\mathcal{I}$ of the signature of O that satisfies all of its axioms.

[4] The signature of an ontology is the set of syntactic elements (*e.g.*, classes, properties, individuals) used in O.

[5] In order to differentiate the interpretations of (non-modular) ontologies and the interpretations of modules, we use the term "local" when we interpret the ontological (non-modular) content of a module, thus the indice l in the satisfaction relation $\models_l$.

4.2 Alignment semantics

An alignment connects entities from 2 different ontologies. Interpreting them implies interrelating both ontology interpretations. In the literature, there are several ways of expressing correspondences in alignment. They can be plain ontological axioms, bridge rules, queries, or $\mathcal{E}$-connections.

Nevertheless, all these formalisms can be describe with the definition given in §3.2 with a common "meta-semantics". Relation symbols $r \in \mathfrak{R}$ appearing in the correspondences are associated to a binary relation $\widetilde{r}$, of which definition is specified by the modular ontology language used.

Definition 3 (Satisfied correspondence). *Let $c = \langle e_1, e_2, r\rangle$ be a correspondence in an alignment between O_1 and O_2. If $\mathcal{I}_1$ and $\mathcal{I}_2$ are interpretations of O_1 and O_2 respectively, then $\langle \mathcal{I}_1, \mathcal{I}_2\rangle$ is said to satisfy c iff $e_1^{\mathcal{I}_1}\ \widetilde{r}\ e_2^{\mathcal{I}_2}$. This is written $\mathcal{I}_1, \mathcal{I}_2 \models c$.*

For instance, consider an alignment language where relation symbol $\sqsubseteq$ is associated to the inclusion of sets, *i.e.*, $\widetilde{\sqsubseteq}$ is $\subseteq$. In this case, $\langle e_1, e_2, \sqsubseteq\rangle$ is satisfied iff $e_1^{\mathcal{I}_1} \subseteq e_2^{\mathcal{I}_2}$.

Definition 4 (Model of an alignment). *Given two ontologies O_1 and O_2 and an alignment A of O_1 and O_2, a model of A is a pair $\langle \mathcal{I}_1, \mathcal{I}_2\rangle$ such that for all $c \in A$, $\mathcal{I}_1, \mathcal{I}_2 \models c$. The set of all models of A is written* $\mathrm{Mod}(A)$.

With this definition, the models of an alignment do not have to satisfy the ontologies. This is useful when one needs to determine consistency of an alignment, but do not have access to the ontologies, or when for other alignment manipulations. Moreover, this also ensures encapsulation at the alignment level, since it prevents alignment satisfiability to be dependent on a particular ontology implementation. Sect. 5 describes how existing alignment formalisms comply with these abstract definitions and discusses their advantages and disadvantages.

With the notable exception of $\mathcal{E}$-connection, all these formalism can be adopted to provide a formal semantics to alignments in our module framework. The next section describe the global module semantics.

4.3 Interpreting modules

In the most general case, the interpretation of a module is recursively defined in function of the interpretations of its imported modules. This recursive definition assumes that there is no cycle in the import chain, so each chain eventually leads to a base module with no import. Detection of cycles should be syntactically checked, since this definition is not well founded otherwise. If one thinks in term of software engineering, this is not a major limitation. Indeed, when a new module is designed, it has to import existing modules. This way, it is not possible to have cyclic references.

Definition 5 (Base module interpretation). *Let $\mathfrak{M} = \langle \mathrm{id}, \emptyset, \emptyset, \emptyset, O, E\rangle$ be a base module. An interpretation of $\mathfrak{M}$ is a local interpretation $\mathcal{I}$ of the content O of $\mathfrak{M}$, with domain of interpretation $\mathcal{D}$.*

Definition 6 (Module interpretation). *Let $\mathfrak{M} = \langle \mathrm{id}, M, I, A, O, E \rangle$ be a module. An interpretation of $\mathfrak{M}$ is a triple $\mathfrak{I} = \langle \mathcal{I}, (\mathcal{I}_m)_{m \in M}, \widetilde{\cdot} \rangle$ such that:*

- *$\mathcal{I}$ is an (local) interpretation of the content O of $\mathfrak{M}$, with domain of interpretation $\mathcal{D}$. $\mathcal{D}$ is also called the* domain of interpretation *of module $\mathfrak{M}$;*
- *For each imported module $m \in M$, $\mathcal{I}_m$ is a* module interpretation *of m over domain of interpretation $\mathcal{D}_m$;*
- *$\widetilde{\cdot}$ is a mapping that associates to each $r \in \mathfrak{R}$ a binary relation $\widetilde{r}$ according to the corresponding semantics of alignments (see §5.1 for specific alignment semantics).*

In order for an interpretation to satisfy a module, there are four conditions:

1. the local interpretation must be a model of the content of the module, *i.e.*, all local axioms must be satisfied;
2. the imported modules must be satisfied by their respective interpretations;
3. the alignments between the imports must be satisfied by the respective pairs of interpretations;
4. finally, the local interpretation and the imported modules interpretation must agree on the interpretation of the interface I.

The fourth item is a bit ambiguous. The notion of "agreement" on the interpretation actually depends on the specific formalism used. This is discussed in the next section.

In order to reason with modular ontologies, we have to define what are the semantic consequences of a module. They are defined as follows:

Definition 7 (Consequences of a module). *Let $\mathfrak{M} = \langle \mathrm{id}, M, I, A, O, E \rangle$ be a module. Let δ be an axiom built upon the signature of the content of $\mathfrak{M}$ (which includes the import interfaces of I). δ is a consequence of $\mathfrak{M}$, written $\mathfrak{M} \models \delta$ iff for all models $\langle \mathcal{I}, (\mathcal{I}_m)_{m \in M}, \widetilde{\cdot} \rangle$ of $\mathfrak{M}$, $\mathcal{I} \models \delta$.*

Obviously, if a formula is a consequence of the content ontology of a module, then it is a consequence of the module itself.[6] Additionally, it is desirable to derive knowledge about the imported terms according to the imported modules knowledge. However, if something is true about a concept C in a module, it is not necessarily true in another module that imports C.

Depending on the formalism used for module semantics, knowledge that is transfered from imported to importing modules varies. The next section describe how candidate semantics fits within our module framework, and discuss the relevance of each existing formalism.

5 Discussion and related work

Work related to modular ontologies has expended greatly in recent years. We will divide this section into two parts. The first focuses on logical formalisms developed to reason with modular ontologies. We show that our modular framework can be adapted to most of them, but some better guarantee the properties that we advocate. The second section presents other related work.

[6] Note that only the consequences related to the exported terms are useful to an external module that imports them.

5.1 Modular ontology languages

An already deep and interesting discussion about modular ontology languages have been carried on in [8]. However, our vision of ontology modularization differs from the authors', and we do not share some of the assumption they make.

Modular ontologies without ontology languages: It is possible to implement modules without any specific formal language dedicated to it. [8] proves that reasoning in P-DL or in a restriction of DDL and of $\mathcal{E}$-connection is logically equivalent to questioning a non-standard reasoning service over a traditional description logic. However, this completely discard the engineering approach. Indeed, although the execution of an encapsulated function is equivalent to the execution of the same inlined function, the utility of encapsulation is unquestioned.

Morevoer, in [7], the authors define two such reasoning services (conservative extension and locality) that are pretended to be necessary to achieve modularity. Unfortunately, it is almost impossible to guarantee conservative extensions when modules have been developed independently and are related with alignments.

Nonetheless, an advantage of this approach is that the consequences of a module relative to its export are correctly transfered to the modules that import it.

Modularity without alignments: This is the approach offered by P-DL [1]. In fact, modules are only related to each others thanks to "import statements". Then imported terms are directly used in axioms. In fact, this can perfectly fit in our modularization framework, since a correspondence $\langle e_1, e_2, R\rangle$ can represent an axiom $e_1\, R\, e_2$, where R can be $\sqsubseteq$, $\sqsupseteq$, $\equiv$ or other semantic relationships like individual membership $\in$, disjunction $\perp$ and so on. This way, the binary relation $\widetilde{R}$ has the same value as its interpretation in the local representation language. Moreover, P-DL already offers most of the facilities that we expect from a modular system. It is possible to assert that a term is *private* or *public*; imports are explicit. Nonetheless, the lack of a looser alignment definition imposes strong interdependencies between modules. The use of heterogeneous modules can easily break consistency, because modules may have been developed for different context (this is discussed in [2]).

Pan et al. [13] define a notion of semantic import that differs from P-DL. The interpretation of imported terms do not coincide entirely between the imported and importing module. The local interpretation of an imported term must be equal to the import's interpretation of the term intersected with the local domain of interpretation. For instance, consider a class C_j defined in module j, which is imported by module i. Then, the local interpretation $\mathcal{I}_i$ of i and $\mathcal{I}_j$ of j has to satisfy $C_j^{\mathcal{I}_i} = C_j^{\mathcal{I}_j} \cap \Delta^{\mathcal{I}_i}$, where $\Delta^{\mathcal{I}_i}$ is the domain of interpretation $\mathcal{I}_i$. There is a sound and complete procedure to determine whether an axiom is transfered through the import chain. This offer an alternative import semantic that we have to consider further.

Using bridge rules: Bridge rules were introduced with DDL [2]. These rules express a semantic relation asserted to be true from one module point of view. A correspondence

$\langle e_1, e_2, R\rangle$ represents a bridge rule $1:e_1 \xrightarrow{R} 1:e_2$, where R may be $\sqsubseteq$, $\sqsupseteq$, $=$ in DDL or $\perp$, $*$ in C-OWL. The relations $\widetilde{R}$ are defined as follows: there exists a domain relation r_{12} s.t. $e_1^{\mathcal{I}_1} \widetilde{\sqsubseteq} e_2^{\mathcal{I}_2}$ (resp. $e_1^{\mathcal{I}_1} \widetilde{\sqsupseteq} e_2^{\mathcal{I}_2}$, $e_1^{\mathcal{I}_1} \widetilde{=} e_2^{\mathcal{I}_2}$, $e_1^{\mathcal{I}_1} \widetilde{\perp} e_2^{\mathcal{I}_2}$, $e_1^{\mathcal{I}_1} \widetilde{*} e_2^{\mathcal{I}_2}$) iff $r_{12}(e_1^{\mathcal{I}_1}) \subseteq e_2^{\mathcal{I}_2}$ (resp. $r_{12}(e_1^{\mathcal{I}_1}) \supseteq e_2^{\mathcal{I}_2}$, $r_{12}(e_1^{\mathcal{I}_1}) = e_2^{\mathcal{I}_2}$, $r_{12}(e_1^{\mathcal{I}_1}) \cup e_2^{\mathcal{I}_2} = \emptyset$, $r_{12}(e_1^{\mathcal{I}_1}) \cup e_2^{\mathcal{I}_2} \neq \emptyset$). Such rules are also used by C-OWL [3] and, with a revisited semantics in [10].

Because of the domain relations, this type of semantics very well comply with heterogeneous knowledge. However, they lead to unintuitive inferences, that are partly solved by [10]. Moreover, bridge rules are not transitive, which forbids alignment composition (as proved in [17]). Since alignment reuse is one of our key concern, we consider it a strong drawback.

A non-directional version of bridge rules is given in [16]. So, this formalism allows alignment composition and offers enough tolerance to heterogeneity. However, no complete reasoning procedure nor complexity results are provided by the authors. Nonetheless, it may prove interesting to investigate, and also fits well in our modular framework.

Relating ontologies via queries: In [15], two modules are related with external concept definitions $C_1 \equiv M_2 : Q_2$, where C_1 is a concept name, M_2 is a module and Q_2 is a conjunctive query. This is represented as a correspondence $\langle C_1, Q_2, \equiv\rangle$ and $C_1^{\mathcal{I}_1} \widetilde{\equiv} Q_2^{\mathcal{I}_2}$ means there is a domain relation r_{12} such that $r_{12}(C_1^{\mathcal{I}_1}) = Q_2^{\mathcal{I}_2}$. This approach addresses two important issues in modularization: compilation and change robustness. Though it has not been investigated so far in other approaches (to the best of our knowledge), knowledge compilation could be envisaged with other formalisms. In fact, since the export interface is supposed to stay the same, the knowledge derivable from it could also be maintained and crystallized as precompiled axioms. There are several problems though. External concept definitions are directional in the same sense as bridge rules, so they are not composable. Moreover they only relate a module to its imports, not two imports together. This means that it cannot take advantage of externally defined alignments between two imports.

$\mathcal{E}$-connection: $\mathcal{E}$-connections [11] are quite different from what precedes. They use cross-ontology role restrictions to relate concepts from different modules. In fact, an $\mathcal{E}$-connection assertion can involve terms from more than two ontologies. Therefore, it is not possible to represent these in an alignment as defined in Def. 2. However, $\langle C_1, C_2, R\rangle$ can represent axiom $\langle R\rangle C_1 \equiv C_2$. $\widetilde{R}$ is then defined as follows: there is a binary relation R^M s.t. $R^M(C_1^{\mathcal{I}_1}) = C_2^{\mathcal{I}_2}$. The main advantage of $\mathcal{E}$-connection is the possibility to distribute reasoning, and its strong tolerance to heterogeneity. However, $\mathcal{E}$-connection is difficult to integrate in our framework because cross-ontology axioms are part like the one given above are part of the ontology, while our approach uses ontology alignments that can be manipulated independently from the ontologies.

5.2 Other related work

Since non-modular ontologies already exist, several teams are working on extracting modules out of existing large ontologies [9, 4, 14]. Although this issue is an important

one, these work do not consider the extraction of reusable module nor the design of a reuse-oriented module system, but rather a set of classical OWL ontologies with which reasoning is efficient. Indeed, none of these works is capable of defining a clean interface for encapsulating the content.

Modularity with CASL as described in [12] is, like our approach, designed with no particular logics in mind. It shows how to use the specification language CASL to develop modular ontologies. Although this approach has some similarity with ours, it does not offer support for specifying explicitly what is exported and imported. Moreover, it has no support for ontology alignments. Since it merely focus on the ontology design task, its purpose is orthogonal to ours.

6 Conclusion

Ontology modules are now urgently needed on the semantic web: the lack of support for modules in OWL has to be corrected. Most of the effort has so far concentrated on partitioning ontologies for providing more efficient reasoning, or on studying the semantic properties that have to be ensured by a module system. Real engineering facilities were often overlooked, with notable exceptions.

In this paper we have rather focused on ontology modules from an ontology engineering standpoint. As such, the important properties of modules are encapsulation, information hiding, replaceability of modules and reusability. Moreover, a module system should provide some glue in order to align the interface between imported modules or imported and importing modules.

From this perspective, we designed a module system reusing ontologies and ontology alignments. This ensures a flexible accommodation of existing external ontology alignments. We provided a syntax allowing the specification of what is imported and exported by a module, while not being stuck to a particular logical formalism. The concrete XML syntax is grounded on OWL ontologies and the Alignment format.

An important issue is to define what are the consequences of a module in the same way the consequences of an ontology are defined. We showed how to accommodate existing modular ontology formalism in order to reason with modules. Unfortunately, we showed that none of them gathers all the properties expected from a modular system. Yet, our framework is general enough to capture many ontology and ontology alignment languages, and considers alignments as distinct objects that can be manipulated separately.

According to this specification, we are currently developing a module system able to fully take advantage of existing OWL ontologies and alignments. In the near future, we will integrate additional facilities for the export interface, that will allow to guarantee by proof that a property holds for the exported terms (like invariance of the concept hierarchy).

References

1. Jie Bao, Doina Caragea, and Vasant Honavar. On the Semantics of Linking and Importing in Modular Ontologies. In *Proc. of 5th International Semantic Web Conference (ISWC'06)*, volume 4273 of *LNCS*, pages 72–86. Springer, 2006.

2. Alex Borgida and Luciano Serafini. Distributed Description Logics: Directed domain correspondences in federated information sources. In *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE : Confederated International Conferences CoopIS, DOA, and ODBASE 2002*, volume 2519 of *LNCS*. Springer, 2002.
3. Paolo Bouquet, Fausto Giunchiglia, Frank van Harmelen, Luciano Serafini, and Heiner Stuckenschmidt. C-OWL: Contextualizing ontologies. In *Proc. 2nd International Semantic Web Conference (ISWC'03)*, volume 2870 of *LNCS*. Springer, 2003.
4. Mathieu d'Aquin, Marta Sabou, and Enrico Motta. Modularization: a Key for the Dynamic Selection of Relevant Knowledge Components. In *Proc. 1st Workshop on Modular Ontologies (WoMO'06)*, 2006.
5. Jérôme Euzenat. An API for Ontology Alignment. In *Proc. of 3rd International Semantic Web Conference (ISWC'04)*, volume 3298 of *LNCS*, pages 698–712. Springer, 2004.
6. Jérôme Euzenat and Pavel Shvaiko. *Ontology matching*. Springer, Heidelberg (DE), 2007.
7. Bernardo Cuenca Grau, Ian Horrocks, Yevgeny Kazakov, and Ulrike Sattler. A logical framework for modularity of ontologies. In *Proc. of 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 298–303, 2007.
8. Bernardo Cuenca Grau and Oliver Kutz. Modular ontology languages revisited. In *Workshop on Semantic Web for Collaborative Knowledge Acquisition (SWeCKa'2007*, 2007.
9. Bernardo Cuenca Grau, Bijan Parsia, Evren Sirin, and Aditya Kalyanpur. Modularity and web ontologies. In *Proc. 10th International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*, pages 198–209. AAAI Press, 2006.
10. Martin Homola. Distributed Description Logics Revisited. In *Proc. of the 20th International Workshop on Description Logics DL'07*. Bolzano University Press, 2007.
11. Oliver Kutz, Carsten Lutz, Frank Wolter, and Michael Zakharyaschev. $\mathcal{E}$-connections of abstract description systems. *Artificial intelligence*, 156(1):1–73, 2004.
12. Klaus Lüttich, Claudio Masolo, and Stefano Borgo. Development of modular ontologies in casl. In *Proc. 1st Workshop on Modular Ontologies (WoMO'06)*, 2006.
13. Jeff Pan, Luciano Serafini, and Yuting Zhao. Semantic Import: an Approach for Partial Ontology Reuse. In *Proc. 1st Workshop on Modular Ontologies (WoMO'06)*, 2006.
14. Julian Seidenberg and Alan Rector. Web ontology segmentation: analysis, classification and use. In *Proc. 15th World Wide Web Conference (WWW'06)*, pages 13–22, 2006.
15. Heiner Stuckenschmidt and Michel Klein. Integrity and Change in Modular Ontologies. In *Proc. 18th International Joint Conference in Articifial Intelligence (IJCAI'03)*, pages 900–908. Morgan Kaufmann, 2003.
16. Antoine Zimmermann. Integrated Distributed Description Logics. In *Proc. of the 20th International Workshop on Description Logics DL'07*. Bolzano University Press, 2007.
17. Antoine Zimmermann and Jérôme Euzenat. Three Semantics for Distributed Systems and their Relations with Alignment Composition. In *Proc. of 5th International Semantic Web Conference (ISWC'06)*, volume 4273 of *LNCS*, pages 16–29. Springer, 2006.