Improved Algorithms for Module Extraction and Atomic Decomposition

Dmitry Tsarkov

tsarkov@cs.man.ac.uk
School of Computer Science
The University of Manchester
Manchester, UK

Abstract. In recent years modules have frequently been used for ontology development and understanding. This happens because a module captures all the knowledge an ontology contains in a given area, and often is much smaller than the whole ontology. One useful modularisation technique for expressive ontology languages is locality-based modularisation, which allows for fast (polynomial) extraction of modules.
In order to better understand the modular structure of an ontology, a technique called Atomic Decomposition can be used. It efficiently builds a structure representing all possible modules for an ontology, regardless of the modularisation algorithm adopted and without the need to compute an exponential number of modules, as in a naive approach. This structure may be used e.g., for quick extraction of modules, or to investigate dependencies between modules, and so on.
However, existing algorithms for both locality-based module extraction and atomic decomposition do not scale well. This happens mainly because of their global nature: each iteration always explores the whole ontology, even when it is not necessary.
We propose algorithms for locality-based module extraction and atomic decomposition that work only on the relevant part of the ontology. This improves performance of algorithms by avoiding unnecessary checks. Empirical evaluation confirms a significant speed up on real-life ontologies.

1 Introduction

Following the great success of the OWL 2 family of ontology languages, they have become widespread as a knowledge representation formalism. This success, in particular, is based on reasoning facilities available for these languages, that are provided by a number of tools. However, the tools (usually) do not scale well. The worst-case computational complexity of the most expressive decidable fragment of OWL, OWL 2 DL, is N2EXPTIME. So there is a need to deal with large ontologies in an efficient manner.

One way of dealing with this issue during ontology development is to use *modules*. A module is a subset of an ontology that captures all the knowledge the ontology contains about a given set of terms. When reusing an existing ontology, instead of importing all of it to use a few terms and axioms, one could

extract a module based on a given set of terms, thus limiting the growth of the ontology that will be fed to the reasoner.

To do so, a module extraction algorithm is necessary. The notion of *conservative extensions* [3] allows one to define a module w.r.t. a signature Σ as a minimal set that preserves all entailments over Σ. However, deciding whether a subset of an ontology is a module in this sense is a non-trivial task. Even for simple DL languages, it is double exponential in time, whereas for expressive languages, like OWL 2 DL, this problem is undecidable.

Locality-based modules is an alternative solution for efficient module extraction in expressive logics. Intuitively, an axiom is *local* w.r.t. a signature if it does not affect any entailment that uses only terms from that signature. So the approach is to keep in the module only those axioms that are non-local to a given signature (while extending the signature as more axioms are added to the module). The traditional modularisation algorithm follows this idea by traversing all the axioms in the ontology, checking their locality and adding them to a module if non-local, updating the signature accordingly and then repeating the traversal until no new entities are added to a signature.

It is easy to see that this approach, while having polynomial (quadratic) run-time, has some room for improvement. The addition of a single term to a signature could lead to re-checking locality of every axiom in the ontology, including those that have nothing to do with the change in the signature, i.e., are not touched by either the old or the new signature. In the approach proposed in this paper only the axioms that might become non-local after a change of signature are re-checked.

There might be cases where one wants to extract more than just a single module from an ontology. In order to explore the modular structure of the ontology, the *atomic decomposition* approach has recently been investigated [1, 5]. Atomic decomposition can be viewed as a compact representation of *all* the modules of an ontology. In this approach the notion of *atom* is introduced as a subset of an ontology, whose axioms always co-occur in a module (i.e., for each module, either all of the axioms are included in the module or none of them occurs in it). A dependency between atoms, which mirrors the subset relation between corresponding modules, is also described in the Atomic Decomposition approach.

The algorithm for building the atomic decomposition of an ontology is also rather straightforward. First, the module for a signature of every axiom is built. Then axioms with equivalent modules are combined into a single atom. After the set of atoms is known, their modules are explored to derive dependencies between atoms.

As in the module extraction case, the atomic decomposition algorithm can be improved. Taking account of the notion of a module in the atomic decomposition structure, it is possible to significantly reduce the search space for modules, as well as for dependency relation. Empirical evaluation on large real-life ontologies shows an increase in performance of up to 50 times.

The rest of this paper is organised as follows. In Section 2 some preliminary notions are defined. Section 3 contains the definition of the original module extraction algorithm, the analysis of its inefficiencies and the improved algorithm, together with a proof of its correctness. Similarly, in Section 4 the original and improved algorithms for the atomic decomposition are discussed. Results of the evaluation of the algorithms involving several real-life ontologies are presented in Section 5. Finally some conclusions are drawn in Section 6.

2 Preliminaries

We assume that the reader is familiar with the notion of OWL 2 axiom, ontology and entailments. An *entity* is a named element of the signature of an ontology. For an axiom α, we denote by $\widetilde{\alpha}$ the signature of that axiom, i.e. the set of all entities in α. The same notation is also used for a set of axioms.

Definition 1 (Module). *Let O be an ontology and Σ be a signature. A subset M of the ontology is called a* module *of O w.r.t. Σ if $M \models \alpha \iff O \models \alpha$, for every axiom α with $\widetilde{\alpha} \subseteq \Sigma$.*

One of the ways to build modules is to use *locality* of axioms.

Definition 2 (Semantic Locality). *An axiom α is called* $\top(\bot)$-local *w.r.t a signature Σ if replacing all named entities in $\widetilde{\alpha} \setminus \Sigma$ with $\top$(resp. $\bot$) makes that axiom a tautology. An axiom α is called a* tautology *if it is local w.r.t. $\widetilde{\alpha}$. An axiom α is called* global *if it is non-local w.r.t. $\emptyset$.*

Note that checking the tautology of an axiom α is done by checking the entailment of α by the empty ontology, i.e., $\emptyset \models \alpha$. In order to avoid this check (which involves reasoning and might be expensive) the notion of *syntactic locality* was introduced in [2].

We are not giving a formal definition of syntactic locality here. This would be an unnecessary complication, as the algorithms will use the locality checker as a black box. The intuition behind the syntactic locality is that it tries to simulate the entailment check by exploring the axiom structure and making decisions about locality by propagating constant values through expressions.

Syntactic locality is sound in the sense that every syntactically local axiom is also semantically local. The converse is not true, however: some syntactically non-local axiom are semantically local. We assume that the locality checker provides a method isNonLocal(α) that returns **true** iff the axiom α is non-local.

Definition 3 (Atomic Decomposition). *A set of axioms A is an* atom *of an ontology O, if for every module M of O, either $A \subseteq M$ or $A \cap M = \emptyset$. An atom A is* dependent *on B (written $B \preccurlyeq A$) if $A \subseteq M$ implies $B \subseteq M$, for every module M. An* Atomic Decomposition *of an ontology O is a graph $G = \langle S, \preccurlyeq \rangle$, where S is the set of all atoms of O.*

Algorithm 1 Original Modularity Algorithm [2]

```
1: function getModule(Σ, O)
2:     M ← ∅, Σ0 ← ∅
3:     repeat
4:         Σ0 ← Σ
5:         for α ∈ O do
6:             if α ∉ M and isNonLocal(α, Σ) then
7:                 M ← M ∪ α
8:                 Σ ← Σ ∪ α̃
9:             end if
10:        end for
11:    until Σ ≠ Σ0
12:    return M
13: end function
```

3 Module Extraction Algorithms

The locality-based module extraction is based on the following theorem.

Theorem 1 (Locality-based Module [2]). *Let $M \subseteq O$ be two ontologies such that all axioms in $O \setminus M$ are local w.r.t. $\Sigma \cup \widetilde{M}$. Then M is a module of O w.r.t Σ*

This claim holds for all types of modules, as well as the locality checking approach. The original algorithm, based on this theorem, is described here as an Algorithm 1, and is implemented in, e.g., OWL API.[1]

3.1 Original Module Extraction Algorithm

The algorithm starts from the empty module and then goes through all the axioms to check their locality. If an axiom α is non-local w.r.t. the current signature, it is added to the module. In addition, the signature is extended with the signature of α. The whole process is repeated until the signature reaches a fixpoint (line 11).

While having a simple structure and being easily understandable, the traditional algorithm has some inefficiencies. The most obvious one comes from the fact that it is necessary to check all the remaining axioms in the ontology if a single entity is added to the signature. This leads to the worst-case complexity of $O(n^2)$, where n is the number of axioms in the ontology. Indeed, if every run of the loop in lines 3–11 adds a single axiom to a module, increasing the signature on each step, the loop will be run n times and about $n^2/2$ locality checks will be made.

[1] http://owlapi.sourceforge.org

Algorithm 2 Improved Modularity Algorithm

```
1: function getModule(Σ, O)
2:     SA ← ∅, Globals ← ∅, M ← ∅
3:     for all α ∈ O do                          ▷ Initialize SA and Globals
4:         if isNonLocal(α, ∅) then              ▷ global axiom
5:             Globals ← Globals ∪ {α}
6:         else
7:             for all σ ∈ α̃ do
8:                 SA(σ) ← SA(σ) ∪ {α}
9:             end for
10:        end if
11:    end for
12:    S ← Σ                                     ▷ Initialise working set
13:    for all γ ∈ Globals do                    ▷ Global axioms are always in the module
14:        addNonLocal(γ, Σ, M, S)
15:    end for
16:    for all σ ∈ S do
17:        S = S \ {σ}
18:        for all α ∈ SA(σ) do
19:            addNonLocal(α, Σ, M, S)
20:        end for
21:    end for
22:    return M
23: end function

24: procedure addNonLocal(α, Σ, M, S)
25:    if α ∉ M and isNonLocal(α, Σ) then
26:        M ← M ∪ α
27:        S ← S ∪ (α̃ \ Σ)
28:        Σ ← Σ ∪ α̃
29:    end if
30: end procedure
```

3.2 Improved Modularity Algorithm

The approach we propose in this paper replaces the global search over all axioms in the ontology with a search over a reduced set of possibly affected axioms. When a new entity is added to the signature, the algorithm checks locality only of the axioms that contain this entity in their signature. This is correct due to the following fact:

Proposition 1. *Let Σ be a signature, and α an axiom such that α is local w.r.t. Σ. Then α is also local w.r.t. any signature $\Sigma \cup \Sigma'$ such that $\Sigma' \cap \widetilde{\alpha} = \emptyset$.*

Proof. Let $\alpha|^{c}_{\Sigma}$ denote the axiom α in which all entities not in Σ are replaced with c, where c is either $\top$ or $\bot$ depending on the locality type. Then the claim of the proposition follows from two simple observations:

1. $\alpha|^{c}_{\Sigma_1 \cup \Sigma_2} = (\alpha|^{c}_{\Sigma_1})|^{c}_{\Sigma_2}$

2. $\alpha|^c_{\Sigma\setminus\widetilde{\alpha}} = \alpha$

Since α is local w.r.t. Σ, $\alpha|^c_\Sigma$ is a tautology. The, by the first item above, $\alpha|^c_{\Sigma\cup\Sigma'} = (\alpha|^c_\Sigma)|^c_{\Sigma'} = (\alpha|^c_\Sigma)|^c_{\Sigma'\setminus\widetilde{\alpha}}$, which, by the second item, equals to $\alpha|^c_\Sigma$. So, $\alpha|^c_{\Sigma\cup\Sigma'}$ coincides with $\alpha|^c_\Sigma$, i.e., is a tautology. Thus, α is local w.r.t. $\Sigma\cup\Sigma'$. □

Algorithm 2 implements the proposed approach. In lines 3–11, the auxiliary structures for the algorithms are initialised. One of these structures is a map SA that associates each entity with the set of axioms containing it in their signature. Another is a set of global axioms *Globals*. Global axioms should be treated in a special way as they are part of every module independently of their signature.

After these structures are created, the algorithm initialises the working set S with the initial signature Σ. Then, all the global axioms from the set *Globals* are added to the module using the addNonLocal procedure.

In the main cycle (lines 16–21) an entity σ is taken from the set S, then the set of affected axioms is retrieved using the map SA. Each of these axioms is checked for locality and, if non-local, is added to the module by addNonLocal.

The addNonLocal procedure is defined in the lines 24–30 of the Algorithm 2. If an axiom is found non-local, it is added to the module M, and its signature is added to Σ. Moreover, every new entity is added to the working set S (line 27) to allow further search for the axioms that are non-local w.r.t. the extended signature.

The correctness of the algorithm can be proved by induction on the size of the signature Σ. The basis of induction, for the empty signature: all that goes to the module is the set of global axioms, which is done in the lines 13–15 of the algorithm. Assume that for a given Σ all the necessary locality checks have been performed for axioms in the ontology O. Let us now check the case when a new entity σ is added to Σ. In this case all the axioms from O that contain Σ in their signature, are re-checked for locality w.r.t. new signature. All other axioms, according to Proposition 1, will keep their locality status, so there is no need to re-check them.

Note that the computational complexity of the improved algorithm is different from th eone of the original one. Now every axiom α is checked an most $|\widetilde{\alpha}|$ times, so the overall complexity is $O(N \times s)$, where N is the size of the ontology O and $s = \max_{\alpha\in O}(|\widetilde{\alpha}|)$.

It is also worth noting that the initialisation of auxiliary structures (lines 3–11) can be done only once for every ontology, and then reused for consequent module extraction queries.

4 Atomic Decomposition Algorithms

Now let us introduce the algorithms for the atomic decomposition of an ontology. For the ease of explanation we assume that the ontology for the atomic decomposition does not contain tautologies (i.e. axioms that are local w.r.t. their own signature). They have no sense in the atomic decomposition, as they does not

Algorithm 3 Original Atomic Decomposition [1]

```
1: procedure AtomicDecomp(O)
2:     Gen ← ∅, Module ← ∅, Atom ← ∅, ≼← ∅
3:     for all α ∈ O do                                  ▷ build all atoms and modules
4:         Module(α) ← getModule(α̃, O)
5:         if isNewModule(α, Gen) then
6:             Atom(α) ← {α}
7:             Gen ← Gen ∪ α
8:         end if
9:     end for
10:    for all α ∈ Gen do                                ▷ build all dependencies
11:        for all β ∈ Gen do
12:            if α ∈ Module(β) then
13:                ≼←≼ ∪⟨Atom(α), Atom(β)⟩
14:            end if
15:        end for
16:    end for
17: end procedure
18: function isNewModule(α, Gen)
19:    for all β ∈ Gen do
20:        if Module(α) = Module(β) then
21:            Atom(β) ← Atom(β) ∪ {α}
22:            return false
23:        end if
24:        return true
25:    end for
26: end function
```

belong to any (locality-based) module of the ontology. In order to achieve this one have to check all the axioms and remove the tautologies from the ontology.

4.1 The Original Atomic Decomposition Algorithm

The original atomic decomposition algorithm presented here was described by Del Vescovo et al [1]. It contains two independent parts. The first part (lines 3–9) builds all the atoms of the ontology. It is done by creating a module for a signature of every axiom α (line 4), and by comparing this module to already created modules. If such a module already exists in the ontology (which is checked in the auxiliary procedure isNewModule, line 20), then the axiom is added to the atom, represented by the already checked axiom β (line 21). If no module is equivalent to the module for α, then α goes to a new atom (line 6).

The second part of the algorithm, lines 10–16, builds the dependency relation $\preccurlyeq$. It goes through all atoms and sets the dependency between atoms A and B if an axiom from A is contained in the module built for axioms from the atom B.

4.2 Improved Atomic Decomposition Algorithm

As in the case of the module extraction algorithm, the traditional atomic decomposition algorithm suffers from some inefficiencies. The first is independence of module creation: all the modules are created, using the whole ontology as a starting point. However, in many cases a module is included into another module with a bigger signature. This is a consequence of the following observation.

Algorithm 4 Improved Atomic Decomposition

Require: An ontology O
Ensure: Set to atoms $Atom$, set of modules $Module$, dependency function $\preccurlyeq$

```
 1: procedure BuildAD(O)
 2:     for all α ∈ O do
 3:         if Atom(α) = ∅ then
 4:             buildAtomsInModule(α, null)          ▷ Set Module(null) to be O
 5:         end if
 6:     end for
 7:     TransitiveClose(≼)
 8: end procedure

 9: function buildAtomsInModule(α, β)
10:     if Atom(α) ≠ ∅ then                          ▷ The atom for α is already known
11:         return α
12:     end if
13:     δ ← getAtomSeed(α, β)
14:     Atom(δ) ← Atom(δ) ∪ {α}
15:     if δ = β then
16:         return β
17:     end if
18:     for all γ ∈ Module(α) \ {α} do
19:         δ ← buildAtomsInModule(γ, α)
20:         ≼ ← ≼ ∪⟨Atom(δ), Atom(α)⟩
21:     end for
22:     return α
23: end function

24: function getAtomSeed(α, β)
25:     Module(α) ← getModule(α̃, Module(β))
26:     if Module(α) = Module(β) then
27:         return β
28:     else
29:         return α
30:     end if
31: end function
```

Proposition 2. *Let α, β be axioms in an ontology O. Let $Module(\gamma, O)$ denote the module of O w.r.t. $\widetilde{\gamma}$. Then $Module(\beta, O) \subseteq Module(\alpha, O)$ whenever $\beta \in Module(\alpha, O)$.*

In fact, this proposition follows from *depleteness* of the locality-based modules [1, Proposition 2.2, claim iii]. So in order to build a module for β it is enough to explore only axioms in the module for α.

Another observation stems from the analysis of an dependency relation structure. Lines 12–13 of the Algorithm 3 imply that all atoms on which $Atom(\beta)$ depends are contained in $Module(\beta)$. But in this case there is no need to look outside that module for the dependencies. At the same time, the dependency relation could be build during the atom creation process.

These two ideas lie at the foundation of the improved atomic decomposition algorithm, presented as Algorithm 4. The main cycle (lines 2–6) ensures that an atom is built for every axiom, using the whole ontology as a starting point. After all atoms are created, the dependency relation is completed by using the standard transitive closure algorithm (line 7).

The main ingredient of the algorithm is implemented as a recursive function buildAtomsInModule. It takes two parameters: an axiom α, for which the atom (and module) are going to be built; and an axiom β, which is a "parent" of an α in the sense that $Module(\alpha) \subseteq Module(\beta)$. For special case $\beta = null$, as in line 4 of the code, we assume that $Module(\beta)$ is the whole ontology O. The function returns a representative of the $Atom(\alpha)$.

First, it checks whether an atom for α has been already created (lines 10–12). In this case there is nothing to do and α is returned as a representative.

Then, using the module of a parent axiom as a starting point, the module for α is created (line 25). Then, like in the function isNewModule from Algorithm 3, the algorithm checks whether such module already exists. However, unlike in function isNewModule, only one check is required here (line 26), namely, to compare it with the parent module. Then axiom α is added to an atom, obtained by function getAtomSeed (line 14) and, if the atom already exists (i.e., is represented by the parent axiom β) then the parent is returned.

If the atom is new, i.e. $Module(\alpha) \neq Module(\beta)$, the algorithm recursively builds all atoms inside $Module(\alpha)$ (lines 18–21): buildAtomsInModule is called for all axioms in $Module(\alpha)$ with α as a parent. The dependency relation is updated accordingly (line 20), as $Atom(\alpha)$ depends on every atom in the $Module(\alpha)$. In the end, α as a representative of a new module, is returned.

5 Empirical Evaluation

The improved algorithms described in Sections 3.2 and 4.2 were implemented in the FaCT++ Description Logic reasoner [4]. We have done some experiments, which show considerable performance improvement over the original versions of the algorithms.

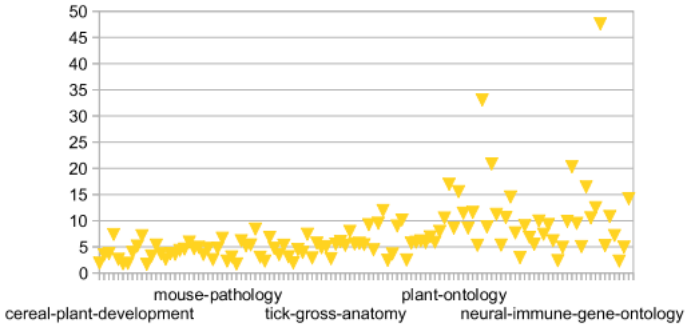
The first set of experiments shows the difference between original and improved module extraction algorithms. As a test we perform an atomic decomposition (improved algorithm) over several well-known ontologies, because it intensively uses the module extraction procedure. The results are presented in Table 1. Here the ontology size is given in the number of axioms, size of the atomic decomposition (AD size) is given in number of atoms.

Table 1. Time and number of locality checks for some ontologies

Ontology	Ont. size,	AD size,	Old algorithm		New Algorithm		Ratio
	#axioms	#atoms	time, sec	nLoc	time, sec	nLoc	
NCI	85,685	54,332	2,282.0	$17.3 \cdot 10^9$	521.2	$330 \cdot 10^6$	52.4
GO	25,117	25,114	414.0	$2.3 \cdot 10^9$	39.6	$88 \cdot 10^6$	26.1
Galen (Full)	4,979	2,699	4.6	$56.7 \cdot 10^6$	2.9	$5.7 \cdot 10^6$	9.9
Wine	869	5	0.0	$125 \cdot 10^3$	0.0	$56 \cdot 10^3$	2.2

This table shows some general patterns of the performance improvement. The first two ontologies represent the case of a large number of small atoms, where the improved algorithm behaves in the best way. The full version of the Galen ontology is very hard for reasoning. It contains one large atom (about 950 axioms), while all other atoms are rather small. Still, the improved algorithm requires only 10% of the locality checks in the original one. The Wine ontology represents the other end of the spectrum: a few very large atoms. This leads to the smallest improvement of the new algorithm against the original one; however, even in this case it uses 50% operations of the standard algorithm.

Fig. 1. Ratio between the improved and original atomic decomposition algorithms on BioPortal ontologies



The second set of experiments compares two atomic decomposition algorithms on a set of ontologies. We use the OWL API implementation as a refer-

ence, and the FaCT++ implementation as an improved algorithm. The set of test ontologies is a subset of BioPortal ontologies, described in [5].

The results of the tests are shown at Fig. 1. The graph shows the ratio between the time needed to decompose ontologies by the original algorithm and the improved algorithm. While the average ratio is about 7, in the extreme cases the improved algorithm demonstrates 48 times better performance.

6 Conclusions

We propose new improved algorithms of the locality-based module extraction and atomic decomposition of the ontologies. We prove their correctness and compare them with the original algorithms. Provided empirical evaluation results confirm that the proposed algorithms outperformed original ones on a set of real-life ontologies.

We are planning to implement a semantic locality checker and compare results on real-life ontologies. We also are planning to implement *labelled atomic decomposition* [5], which could be useful for the fast module extraction.

References

1. Del Vescovo, C., Parsia, B., Sattler, U., Schneider, T.: The modular structure of an ontology: atomic decomposition. In: Proc. of IJCAI. pp. 2232–2237 (2011)
2. Grau, B., Horrocks, I., Kazakov, Y., Sattler, U.: Modular reuse of ontologies: Theory and practice. Journal of Artificial Intelligence Research 31(1), 273–318 (2008)
3. Konev, B., Lutz, C., Walther, D., Wolter, F.: Formal properties of modularisation. Modular Ontologies pp. 25–66 (2009)
4. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. Automated Reasoning pp. 292–297 (2006)
5. Vescovo, C.D., Gessler, D., Klinov, P., Parsia, B., Sattler, U., Schneider, T., Winget, A.: Decomposition and modular structure of bioportal ontologies. In: International Semantic Web Conference (1). pp. 130–145 (2011)