Instrumenting the Atomic Decomposition: software APIs for OWL

Dmitry Tsarkov, Chiara Del Vescovo, and Ignazio Palmisano

School of Computer Science, The University of Manchester, UK

Abstract. The Atomic Decomposition (AD) of an ontology $\mathcal{O}$ is a compact representation of all the modules of $\mathcal{O}$. Besides its theoretical value, it has some practical applications, which include optimised reasoning techniques. In this paper we describe the existing programming APIs that allow for the use of the AD of an ontology in Java and C++ programs.

1 Introduction

Some notable examples of ontologies describe large and loosely connected domains. This is the case for SNOMED–CT, the Systematized Nomenclature Of MEDicine, Clinical Terms,[1] which describes more than 300,000 terms used in medicine, including diseases, drugs, etc. Hence, users often are not interested in a whole ontology, rather only in a limited relevant part of it. In this context, the idea has been recently explored to use *modules*, i.e., suitably small subsets of ontologies that behave for specific purposes as the original ontologies over a given *signature* Σ, i.e., a set of terms.

In reuse scenarios, for example, users may want to gather all the knowlege captured in well-established, comprehensive ontologies, that concerns a "topic of interest" specfied by a user by means of a signature Σ. In this field, a family of widely used modules arises from the notion of *syntactic locality* [2]: syntactic locality allows for the efficient identification in an ontology $\mathcal{O}$ of "good sized" sets of axioms of $\mathcal{O}$ that preserve *all* the entailments over Σ. From now on we will use the term "module" to mean "a module based on syntactic locality".

In general, a module $\mathcal{M}$ can still describe loosely connected areas of the domain, i.e., $\mathcal{M}$ can be decomposed into logically unrelated parts. This is not the case for *genuine modules*, introduced in [4] and defined as modules that cannot be decomposed into the union of two or more modules. The genuine modules of an ontology $\mathcal{O}$ are at most as numerous as the axioms of $\mathcal{O}$, can be efficiently extracted, and can be tersely represented by a Directed Acyclic Graph called the *Atomic Decomposition* (AD) of $\mathcal{O}$, which keeps track of the set inclusion relation occurring between the genuine modules.

The AD reveals crucial semantic relations between the axioms of $\mathcal{O}$, and it is currently investigated for being used in several applications, including optimised techniques able to tackle reasoning tasks over huge and complex ontologies [6].

This paper aims at giving a brief introduction and underlying intuitions behind the Atomic Decomposition, and at describing the implementation and methods for the use of the AD of an ontology in Java and C++ programs available in the OWL API Tools[2].

[1] `http://www.ihtsdo.org/snomed-ct/`

[2] `http://owlapitools.sourceforge.net/`

2 Theoretical Background

We assume the reader to be familiar with OWL and the underlying Description Logics [1], and only briefly sketch here some of the central intuitions behind locality-based modularity [2] and Atomic Decompositions [4]. As seen in Section 1, we use $\mathcal{O}$ for ontologies, i.e. finite sets of OWL axioms, $\mathcal{M} \subseteq \mathcal{O}$ for subsets thereof, and Σ for signatures, i.e., finite sets of class, property, and individual names. Moreover, we use $\widetilde{\alpha}$ for the signature of an axiom α, i.e., the set of terms used in α, and given an ontology $\mathcal{O}$ we denote by $\widetilde{\mathcal{O}}$ the signature obtained as the union of the signatures of all the axioms in $\mathcal{O}$.

Given two ontologies $\mathcal{M}, \mathcal{O}$ with $\mathcal{M} \subseteq \mathcal{O}$, and a signature Σ, we say that $\mathcal{M}$ *provides coverage for* Σ if, for each axiom α such that $\widetilde{\alpha} \subseteq \Sigma$, it holds that $\mathcal{M}$ entails α if, and only if, $\mathcal{O}$ does. In this case we say that $\mathcal{M}$ is a *module for* Σ *in* $\mathcal{O}$. Some modules can be efficiently found by making use of the notion of *syntactic locality*, available in the OWL API.[3] The technical details behind this notion go beyond the scope of this paper, so we refer the interested reader to [2].

Locality-based modules come in 3 flavours, namely $\top$, $\bot$, and $\top\bot^*$: roughly speaking, a $\top$-module for Σ gives a view "from above" because it contains all the subclasses of class names in Σ; a $\bot$-module for Σ gives a view "from below" since it contains all the superclasses of class names in Σ; and $\top\bot^*$-modules are the smallest, contained in both the corresponding $\top$- and $\bot$-module, containing all the axioms to imply that two classes in Σ are in the subclass relation, but not necessarily all their sub- or superclasses. Given a module notion $x \in \{\top, \bot, \top\bot^*\}$, we denote by x-mod$(\Sigma, \mathcal{O})$ the x-module of $\mathcal{O}$ w.r.t. Σ. From now on, we will drop the symbol x when the notion of module is clear from the context, or irrelevant.

Let us fix a notion x of modules, and let $\mathcal{F}^x(\mathcal{O})$ be the set of all the x-modules of an ontology $\mathcal{O}$. In general, a module can deal with logically unrelated elements of a domain, for example in SNOMED–CT the user is allowed to extract the module for the signature $\Sigma = \{\mathsf{Heart\_Disease}, \mathsf{Tibia\_Fracture}\}$. In contrast, a special class $\mathfrak{G} \subseteq \mathcal{F}(\mathcal{O})$ of logically coherent modules, called *genuine*, is identified in [4]: these modules can be said to be coherent since they are undecomposable, i.e. none of them can be split into the union of two "$\subseteq$"-uncomparable modules.

For each notion x, the corresponding genuine modules satisfy some additional interesting properties: they define a base for all the modules of $\mathcal{O}$, and can be obtained by extracting the modules for the signatures of each single axiom in $\mathcal{O}$. As a consequence, the number of genuine modules is linear in the number n of axioms of $\mathcal{O}$, whilst the number of modules can even be exponential in n [3]. An optimized quadratic procedure to extract all the genuine modules is described in [5].

Given a notion x of modules, and two axioms $\alpha, \beta \in \mathcal{O}$, the genuine modules of $\mathcal{O}$ reveal two kinds of logical relations between these two axioms [4]:
(1) $\alpha \sim_x \beta$, occurring when, for each genuine module $\mathcal{M} \subseteq \mathcal{O}$, we have that $\alpha \in \mathcal{M}$ if, and only if, $\beta \in \mathcal{M}$. A maximal set $\mathfrak{a} \subseteq \mathcal{O}$ of co-occurring axioms is called an *atom*. In other words, two axioms α and β always co-occur when they are so strongly interrelated that the notion of module is not able to separate them. Please note that each

[3] `http://owlapi.sourceforge.net`

axiom belong to one, and only one, atom of $\mathcal{O}$, i.e., the set of atoms forms a partition of $\mathcal{O}$.
(2) $\alpha >_x \beta$, defined to occur when, for each genuine module $\mathcal{M} \subseteq \mathcal{O}$ such that $\alpha \in \mathcal{M}$, then $\beta \in \mathcal{M}$. This relation can be extended to atoms in the following way: an atom $\mathfrak{a}$ is *dependent* on a distinct atom $\mathfrak{b}$ (written $\mathfrak{a} \succ_x \mathfrak{b}$) if, for each genuine module $\mathcal{M}$ such that $\mathfrak{a} \subseteq \mathcal{M}$, then $\mathfrak{b} \subseteq \mathcal{M}$ holds too. In other words, an atom $\mathfrak{a}$ depends on an atom $\mathfrak{b}$ when $\mathfrak{a}$ "needs" to import $\mathfrak{b}$ to make up to a module of $\mathcal{O}$.

The set of atoms of $\mathcal{O}$ is denoted by $\mathcal{A}$, and the pair $(\mathcal{A}, \succ_x)$ is called the *x-Atomic Decomposition (AD)* of $\mathcal{O}$. The AD of an ontology is a terse representation of the genuine modules of $\mathcal{O}$ since there is a 1-1 correspondence between atoms and genuine modules. In [4] it is proven that the dependence relation $\succ$ on $\mathcal{A}$ is a partial ordering[4] (i.e., transitive, reflexive, and antisymmetric). As a consequence, the pair $(\mathcal{A}, \succ)$ can be represented by means of a DAG, as shown in Example 1.

Example 1. Let us consider the ontology Teetotaller defined as follows:

$\{\alpha_1$: Animal SubClassOf hasHabitat min 1 Thing ,
α_2 : Animal SubClassOf hasGender exactly 1 Thing ,
α_3 : Person SubClassOf Animal ,
α_4 : Vegan EquivalentTo Person and eats only (Vegetable or Mushroom),
α_5 : TeeTotaller EquivalentTo Person and drinks only NonAlcoholicThing ,
α_6 : Student SubClassOf Person and hasHabitat some University ,
α_7 : GraduateStudent EquivalentTo Student and hasDegree some $\{BA, BS\}$,
α_8 : Car SubClassOf Vehicle $\}$

The AD of Teetotaller is shown in Figure 1.

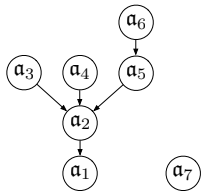

Fig. 1. The $\perp$-AD of the ontology Teetotaller

Here the $\perp$-atoms in the AD contain the following axioms respectively: $\mathfrak{a}_1 = \{\alpha_1, \alpha_2\}$, $\mathfrak{a}_2 = \{\alpha_3\}$, $\mathfrak{a}_3 = \{\alpha_4\}$, $\mathfrak{a}_4 = \{\alpha_5\}$, $\mathfrak{a}_5 = \{\alpha_6\}$, $\mathfrak{a}_6 = \{\alpha_7\}$, and $\mathfrak{a}_7 = \{\alpha_8\}$.

3 Available implementations

The available implementations for AD methods can be found for two programming languages: Java and C++.

[4] This is a consequence of the properties of locality-based modules called monotonicity, self-containment, and uniqueness. The interested reader is referred to [2] for more details.

3.1 Java

The implementation for the Atomic Decomposition available in OWLAPITOOLS[5] is focused on one main interface: `AtomicDecomposition`.

`AtomicDecomposition` provides the fundamental methods for interacting with the AD of an ontology. In particular, given an ontology $\mathcal{O}$ it can perform the following tasks:

- Enumerating the atoms of $\mathcal{O}$;
- Providing the set of the dependent atoms, i.e., the atoms that a given atom is dependent on;
- Providing the set of the dependencies, i.e., the atoms that depend on a given atom;
- Providing the set of the tautologies occurring in $\mathcal{O}$.

These methods allow for performing the basic operations needed to explore a decomposition graph; further operations that a programmer might find useful are:

- Retrieving the atom that includes a specific axiom;
- Retrieving top (i.e., with no dependents) and bottom (i.e., with no dependencies)[6] atoms for the AD graph;
- Retrieving the genuine module corresponding to a given atom;
- Verifying whether an atom $\mathfrak{a}$ is a top or a bottom atom.

Figures 2, 3, and 4 show some examples of Java code that perform the tasks just listed.

```
OWLOntology o = getOntology();
AtomicDecomposition ad = new AtomicDecomposerOWLAPITOOLS(o);
```

Fig. 2. Create an AtomicDecomposition object.

```
for (Atom atom : ad.getAtoms()) {
    for (Atom d : ad.getDependencies(atom)) {
        // do something with the dependencies
    }
    for (Atom d : ad.getDependents(atom)) {
        // do something with the dependents
    }
}
```

Fig. 3. Navigate the graph.

[5] `http://owlapitools.sourceforge.net/`

[6] The terms "top" and "bottom" here do not refer to the notion x of modules used.

```
private void bottomUpExploration(Atom a, Set<Atom> processed) {
    processed.add(a);
    // explore only direct dependencies
    for (Atom dep:ad.getDependencies(a, true)) {
        if (!processed.contains(dep)) {
            bottomUpExploration(dep, processed);
        }
        // real processing of an atom a goes here
    }
}
private void exploreAD() {
    Set<Atom> processed = new HashSet<Atom>();
    for (Atom top:ad.getTopAtoms()) {
        bottomUpExploration(top, processed);
    }
}
```

Fig. 4. Bottom up exploration of the graph.

3.2 C++

The C++ users could use FaCT++ implementation of the Atomic Decomposition algorithms. There are two interfaces for the AD in FaCT++: a low-level one that creates a graph of the AD, and the high-level one, that only exposes a minimal set of operations to explore the AD.

High-level interface The high-level interface is in the class `ReasoningKernel`, file `Kernel/Kernel.h` and consists of 4 methods, that allow one to:

- Create the AD of the ontology loaded into the kernel and get its size;
- Get all the axioms in (the module for) an atom with a given ID;
- Get IDs of all atoms that are (direct) dependents of a given one.

The exact interface is shown in Figure 5.

Low-level Interface The low-level interface gives a user the access to a graph structure representing the AD graph, and allows a user to create the AD graph for a given notion of locality.

The AD graph is represented by a class `AOStructure`, that could be found in the file `AtomicDecomposer.h`. This class is essentially an array of pointers to atoms, that in turn are represented by a class `TOntologyAtom` (file `tOntologyAtom.h`). The access to individual atoms is provided either by means of an iterators, `AOStructure::iterator`, or by an integral index (`operator[]`). The dependency structure of a graph is represented by the `TOntologyAtom` class.

The interface part of the class `TOntologyAtom` allows one to:

```
enum ModuleType = {M_BOT, M_TOP, M_STAR};
class ReasoningKernel {
public:// ...other code
       // create new atomic decomposition of the ontology using TYPE
       // return number of atoms in the AD
    unsigned int getAtomicDecompositionSize
            ( bool useSemanticLocalityChecker, ModuleType Type );
       // get all axioms of the atom with the id ID
    const TOntologyAtom::AxiomSet& getAtomAxioms ( unsigned int id ) const;
       // get all axioms of the module of the atom with the id ID
    const TOntologyAtom::AxiomSet& getAtomModule ( unsigned int id ) const;
       // get all atoms on which atom with id ID directly depends
    const TOntologyAtom::AtomSet& getAtomDependents ( unsigned int id ) const;
};
```

Fig. 5. High-level AD interface in class `ReasoningKernel`.

- get all the axioms in an atom (method `getAtomAxioms`);
- get all the axioms in an atom's genuine module (method `getModule`);
- get the directly dependent atoms (method `getDepAtoms`);
- get all the dependent atoms (method `getAllDepAtoms`);

The construction of an AD graph is performed by the class `AtomicDecomposer`, file `AtomicDecomposer.h`. The only constructor takes as a parameter the module extractor class `TModularizer`, which will be used to extract the genuine modules during the AD creation process.

The creation of the AD is done by calling the method `getAOS` which takes two parameters in input. The first parameter is the ontology to be decomposed; the second is the type of module to be extracted. The supported types are $\bot$, $\top$ and $\top\bot*$, that are denoted by the values `M_BOT`, `M_TOP` and `M_STAR`.

All these low-level details are abstracted to the high-level interface by the `ReasoningKernel` described above.

4 Conclusion

The Atomic Decomposition of an ontology is a useful means to explore all the basic modules of an ontology, and it is used for many applications. In this paper we present the interfaces to create and use the AD of an ontology. One can use Java or C++ programming languages to work with ADs, and in case of C++ one can use different levels of abstractions to work with a representation of the AD.

References

1. Baader, F., Calvanese, Diego andMcGuinness, D., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)

2. Cuenca Grau, B., Horrocks, I., Kazakov, Y., Sattler, U.: Modular reuse of ontologies: Theory and practice. J. of Artif. Intell. Research 31(1), 273–318 (2008)
3. Del Vescovo, C., Parsia, B., Sattler, U., Schneider, T.: The modular structure of an ontology: an empirical study. ceur-ws.org, vol. 573 (2010)
4. Del Vescovo, C., Parsia, B., Sattler, U., Schneider, T.: The modular structure of an ontology: Atomic decomposition. In: Proc. of IJCAI-11. pp. 2232–2237 (2011)
5. Tsarkov, D.: Improved algorithms for module extraction and atomic decomposition. vol. 846. ceur-ws.org (2012)
6. Tsarkov, D., Palmisano, I.: Divide et impera: Metareasoning for large ontologies. ceur-ws.org, vol. 849 (2012)