Undefined 0 (0) 1
IOS Press

HyS: Fast Atomic Decomposition and Module Extraction of OWL-EL ontologies

Francisco Martín-Recuerda [a], and Dirk Walther [b,*]

[a] *Departamento Inteligencia Artificial, Facultad de Informática, Universidad Politécnica de Madrid*
Campus de Montegancedo, 28660 Boadilla del Monte (Madrid) Spain
E-mail: fmartinrecuerda@fi.upm.es
[b] *Theoretical Computer Science, TU Dresden, Germany*
Center for Advancing Electronics Dresden, Germany
E-mail: Dirk.Walther@tu-dresden.de

Abstract. In this paper, we present HyS, an application for fast atomic decomposition and module extraction of OWL-EL ontologies. HyS computes a hypergraph representation of the modular structure of an ontology. This hypergraph representation contains the atomic decomposition of the input ontology, and it allows to extract modules for a given signature. We provide an experimental evaluation of HyS with a selection of large and prominent biomedical ontologies, most of which are available in the NCBO Bioportal.

Keywords: Atomic decomposition, module extraction, syntactic locality, directed hypergraphs, axiom dependency hypergraph

1. Introduction

A *module* is a subset of an ontology that includes all the axioms required to define a set of terms and the relationships between them. Efficient approximations of minimal modules have been suggested based on the notion of *locality* [3]. Module extraction facilitates the reuse of existing ontologies. Some meta-reasoning systems such as MORe[1] and Chainsaw[2] exploit module extraction techniques for improving the performance of reasoning tasks.

The number of all possible modules of an ontology can be exponential wrt. the number of terms or axioms of the ontology [3]. *Atomic decomposition* consists of a polynomial size representation of all possible modules of an ontology. This representation can facilitate the creation, reuse and maintenance of OWL ontologies [5]. Moreover, atomic decomposition can contribute to improve the performance of existing algorithms for locality-based module extraction [4]. Tractable algorithms for computing the atomic decomposition of OWL ontologies have been defined [5], and subsequently improved further [11].

In this paper, we present the application HyS[3] for fast atomic decomposition and module extraction of OWL-EL ontologies. HyS computes an *axiom dependency hypergraph* to represent the modular structure of an ontology [7]. In this hypergraph representation, a locality-based module is equivalent to a *connected* component and an atom to a *strongly connected* component.

We evaluate HyS by comparing it against several state-of-the-art implementations for computing atomic decomposition and module extraction [11,12]. We confirm a significant improvement in running time for a selection of prominent biomedical ontologies from the NCBO Bioportal.

*Supported by the German Research Foundation (DFG) via the Cluster of Excellence 'Center for Advancing Electronics Dresden'.

[1] http://www.cs.ox.ac.uk/isg/tools/MORe/

[2] http://sourceforge.net/projects/chainsaw/

[3] http://lat.inf.tu-dresden.de/~dwalther/software/hys/

The paper is organised as follows. Section 2 provides a brief overview of relevant work on module extraction and atomic decomposition of OWL ontologies. In Section 3 we describe how HyS represents internally an axiom dependency hypergraph and how the atomic decomposition of an ontology and modules are computed. The interfaces of HyS are described in Section 4, and the architecture of HyS in Section 5. In Section 6 we provide an evaluation of HyS. We conclude with a discussion and an overview of future work in Section 7.

2. Related Work

This section provides a brief overview on relevant work on module extraction and atomic decomposition of OWL ontologies. In particular, the notion of syntactic locality is introduced and we also explain how this notion can be used to compute approximations of minimal modules of ontologies. At the end of this section, we present some well-known implementations for computing syntactic locality-based modules and the atomic decomposition of OWL ontologies.

2.1. Syntactic Locality-based Modules

For an ontology $\mathcal{O}$ and a set of symbols (or signature) Σ, a module $\mathcal{M}$ is a subset of $\mathcal{O}$ that preserves all entailments formulated using symbols from Σ only. Computing a minimal module is hard (or even impossible) for expressive fragments of OWL 2. The notion of *syntactic locality* was introduced to allow for efficient computation of approximations of minimal modules [3]. Intuitively, an axiom α is *local* wrt. Σ if it does not state anything about the symbols in Σ. In this case, an ontology defining terms in Σ can *safely* be extended with α, or it can *safely* import α, where 'safe' means not changing the meaning of terms in Σ. A locality-based module wrt. Σ of an ontology consists of the axioms that are non-local wrt. Σ and the axioms that become non-local wrt. Σ extended with the symbols in other non-local axioms. While there are several syntactic locality notions, we focus in this paper only on $\bot$ and $\top$ syntactic locality [3]. We denote with $\mathsf{Mod}^x_{\mathcal{O}}(\Sigma)$ the x-local module of an ontology $\mathcal{O}$ wrt. Σ, where $x \in \{\bot, \top\}$.

Checking for syntactic locality involves checking that an axiom is of a certain form (syntax), no reasoning is needed, and it can be done in polynomial time [3]. However, the state of non-locality of an axiom can also be checked in terms of signature containment [9]. To this end, we introduced the notion of *minimal non-locality signature* of an axiom [7], which is a minimal subset of the axiom signature such that the axiom is not local wrt. this minimal subset. The set of all minimal non-x-locality signatures of an axiom α is denoted by $\mathsf{MLS}^x(\alpha)$.

Figure 1 presents a module extraction algorithm for locality-based modules, where x ranges over any locality notion.

---

function $\mathsf{Mod}^x_{\mathcal{O}}(\Sigma)$ returns x-local module of $\mathcal{O}$ wrt. Σ

1: $m := 0$
2: $\mathcal{M}_m := \emptyset$
3: do
4: $\quad m := m + 1$
5: $\quad \mathcal{M}_m := \{\alpha \in \mathcal{O} \mid \alpha$ is not x-local wrt. $\Sigma \cup \mathsf{sig}(\mathcal{M}_{m-1})\}$
6: until $\mathcal{M}_m = \mathcal{M}_{m-1}$
7: return $\mathcal{M}_m$

Fig. 1. A Module Extraction Algorithm

The algorithm takes an ontology $\mathcal{O}$ and a signature Σ as input, and it produces a finite sequence $\mathcal{M}_0, \ldots, \mathcal{M}_n$, $n \geq 0$, of subsets of the ontology $\mathcal{O}$. Each computed set $\mathcal{M}_i$, $1 \leq i \leq n$, consists of axioms that are not x-local wrt. $\Sigma \cup \mathsf{sig}(\mathcal{M}_{i-1})$ (cf. Line 5), where $\mathsf{sig}(\mathcal{M}_i)$ is the signature of $\mathcal{M}_i$. The algorithm eventually reaches a fixed point (Line 6), and it returns the x-local module of $\mathcal{O}$ wrt. Σ. The algorithm runs in time quadratic in the size of the ontology and signature.

2.2. Atomic Decomposition

An atom is a maximal set of highly related axioms of an ontology in the sense that they always co-occur in modules [5]. Consequently, we have that two axioms α and β are contained in an atom $\mathfrak{a}$ if, and only if, $\mathsf{Mod}^x_{\mathcal{O}}(\mathsf{sig}(\alpha)) = \mathsf{Mod}^x_{\mathcal{O}}(\mathsf{sig}(\beta))$, where $\mathsf{sig}(\alpha)$ and $\mathsf{sig}(\beta)$ are the signatures of the axioms α and β, and $x \in \{\bot, \top\}$. We denote with $\mathsf{Atoms}^x_{\mathcal{O}}$ the set of all atoms of $\mathcal{O}$ wrt. syntactic x-locality modules, where x may range over any locality notion. The atoms of an ontology partition the ontology into pairwise disjoint subsets. All axioms of the ontology are distributed over atoms such that every axiom occurs in exactly one atom.

A dependency relation between atoms can be established as follows. An atom $\mathfrak{a}_2$ depends on an atom $\mathfrak{a}_1$ in an ontology $\mathcal{O}$ if all modules containing $\mathfrak{a}_2$ also contain $\mathfrak{a}_1$. For a given ontology $\mathcal{O}$, the poset $\langle \mathsf{Atoms}^x_{\mathcal{O}}, \succcurlyeq_{\mathcal{O}} \rangle$ was introduced as the *Atomic Decomposition* (*AD*) of $\mathcal{O}$, where $x \in \{\bot, \top\}$. It is of size polynomial in the size of $\mathcal{O}$ [5].

2.3. Relevant Tools

We now introduce some of the most prominent tools that have been optimized to compute syntactic locality-based modules and the atomic decomposition of OWL ontologies.

FaCT++[4] is an OWL 2 reasoner implemented in C++. The latest version, FaCT++ v1.6.3, from May 2014 supports OWLAPI v4.0.1. FaCT++ implements optimized algorithms for computing syntactic locality-based modules and the atomic decomposition of OWL 2 ontologies [11].

OWLAPI-tools[5] is a collection of Java libraries that extend the functionality of the OWLAPI. The latest version, OWLAPI-tools v4.0.1, was released in November 2014 together with the OWLAPI v4.0.1.[6] The collection include a specific library for computing syntactic locality-based modules and the atomic decomposition of OWL 2 ontologies. This library implements the same algorithms as FaCT++ and a similar Java interface [12]. We refer to this library as OWLAPI-AD.

It is also worth mentioning the system CEL[7], a reasoner implemented in LISP for the description logic $\mathcal{EL}^{++}$ [2]. CEL is also capable of computing syntactic locality-based modules for ontologies defined using the description logic $\mathcal{EL}^{++}$. CEL implicitly represents an $\mathcal{EL}^{++}$ ontology as a directed hypergraph where each symbol of the ontology is a node in the hypergraph and each axiom is represented as a directed hyperedge. Connected components in the directed hypergraph correspond to syntactic locality-based modules [9]. The latest version of CEL v1.1.2 was released in August 2009, and the most recent version of the plug-in supports OWLAPI v3.2.4. A version of CEL implemented in Java with the name of jcel[8] is also available. However, jcel does not include the algorithms for computing syntactic locality-based modules.

3. Overview of HyS

HyS is a Java application designed to speed up the computation of syntactic locality-based modules and the atomic decomposition of OWL ontologies. The current version of HyS, version 1.0, supports OWL-EL[9] ontologies and the notion of $\bot$-locality. We plan to extend the implementation to support both, OWL 2 ontologies and $\top$-locality, in the future.

HyS explicitly represents the modular structure of an ontology $\mathcal{O}$ as a directed hypergraph $\mathcal{H}^x_{\mathcal{O}} = (\mathcal{V}, \mathcal{E})$, called *Axiom Dependency Hypergraph (ADH)* [6,7], where $\mathcal{V}$ is the set of nodes, $\mathcal{E}$ is a set of directed hyperedges and $x \in \{\bot, \top\}$ ranges over a syntactic locality notion. The nodes in the ADH correspond to the axioms of $\mathcal{O}$ and each directed hyperedge connects one or more nodes (the *tail* of the hyperedge) with only one node (the *head* of the hyperedge). The tail of a directed hyperedge e represents a minimal set of axioms that provide the signature symbols required by the axiom represented by the head node of e to be non-local. This means that at least one of the minimal non-locality signatures of the axiom represented by the head node of a directed hyperedge e is a subset of the union of the signature symbols of the axiom (or axioms) represented by the tail node (or nodes) of e.

Locality-based modules correspond to connected components in the axiom dependency hypergraph and strongly connected components to atoms [7]. Each strongly connected component of a directed hypergraph can be collapsed into a single node. A *condensed* axiom dependency hypergraph (cADH) is an ADH such that all its strongly connected components have been collapsed into single nodes [7]. Similarly, it is also possible to compute the *partially condensed* axiom dependency hypergraph (pcADH) of an ADH [7]. The idea is to identify and collapse the strongly connected components of the graph fragment of the ADH which has the same nodes as the ADH but only simple directed hyperedges (containing solely one node in the tail). Then, the hyperedges of the ADH are recalculated to consider the newly formed nodes.

[4] http://code.google.com/p/factplusplus/
[5] https://github.com/owlcs/owlapitools/
[6] http://sourceforge.net/projects/owlapi/files/
[7] https://code.google.com/p/cel/
[8] http://jcel.sourceforge.net

[9] OWL-EL profile extended with inverse and functional role axioms

Notice that the number of hyperedges of an ADH may be exponential in the size of the input ontology [6], which makes it impractical to represent the entire ADH explicitly. We implement an ADH $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ as a directed labelled graph $\mathcal{G}_\mathcal{H} = (\mathcal{V}, \mathcal{E}', \mathcal{L})$ containing the simple hyperedges of $\mathcal{H}$ ($\mathcal{E}' \subseteq \mathcal{E}$) and encoding the complex hyperedges in the node labels ($\mathcal{L}$) as follows. A node v_α in $\mathcal{G}_\mathcal{H}$ for an axiom α is labelled with the pair $\mathcal{L}(v_\alpha) = (\mathsf{MLS}^x(\alpha), \mathsf{sig}(\alpha))$ consisting of the minimal non-x-locality signatures of α and the signature of α, where $x \in \{\bot, \top\}$. In fact, not all symbols of $\mathsf{sig}(\alpha)$ are needed in the second component, only those symbols that occur in the minimal non-locality signature of some axiom in the ontology. Condensed (or partially condensed) axiom dependency hypergraphs are implemented in a similar way with the difference that nodes represent sets of axioms.

Figure 2 shows an example of an axiom dependency hypergraph of an ontology $\mathcal{O} = \{\alpha_1, ..., \alpha_5\}$, where each axiom is defined as follows:

- $\alpha_1 = A \sqsubseteq B \sqcap E \sqcap F$
- $\alpha_2 = B \sqsubseteq C \sqcap F$
- $\alpha_3 = B \sqcap C \sqsubseteq D$
- $\alpha_4 = D \sqcap E \sqsubseteq A$
- $\alpha_5 = F \sqsubseteq H$

The axiom dependency hypergraph is as $\mathcal{H}^\bot_\mathcal{O} = (\mathcal{V}, \mathcal{E})$, where:

- $\mathcal{V} = \mathcal{O}$
- $\mathcal{E} = \{e_1 = (\{\alpha_1\}, \alpha_2),\ e_2 = (\{\alpha_2\}, \alpha_3),$
  $e_3 = (\{\alpha_3\}, \alpha_2),\ e_4 = (\{\alpha_1, \alpha_3\}, \alpha_4),$
  $e_5 = (\{\alpha_4\}, \alpha_1),\ e_6 = (\{\alpha_1\}, \alpha_5),$
  $e_7 = (\{\alpha_2\}, \alpha_5)\}$

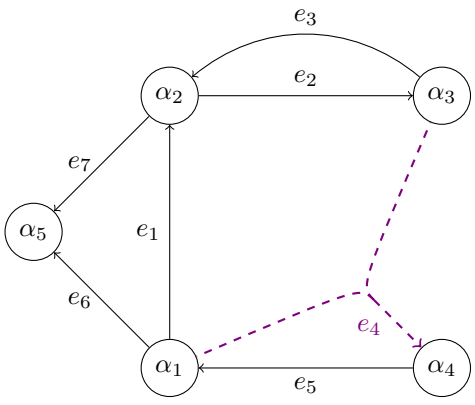

Fig. 2. Axiom dependency hypergraph

Figure 3 represents the directed labelled graph, $\mathcal{G}_\mathcal{H} = (\mathcal{V}, \mathcal{E}', \mathcal{L})$, which encodes the axiom dependency hypergraph, $\mathcal{H}^\bot_\mathcal{O} = (\mathcal{V}, \mathcal{E})$, depicted in Figure 2. The nodes in $\mathcal{G}_\mathcal{H}$ are the same than the nodes in $\mathcal{H}^\bot_\mathcal{O}$. Only simple directed hyperedges are explicitly represented in $\mathcal{G}_\mathcal{H}$. Each node is labelled as follows:

- $\alpha_1 = (\{\{A\}\}, \{A, B, E, F\})$
- $\alpha_2 = (\{\{B\}\}, \{B, C, F\})$
- $\alpha_3 = (\{\{B, C\}\}, \{B, C, D\})$
- $\alpha_4 = (\{\{D, E\}\}, \{A, D, E\})$
- $\alpha_5 = (\{\{F\}\}, \{F, H\})$

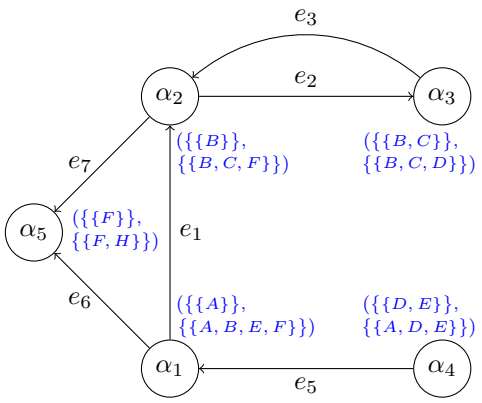

Fig. 3. Directed labelled graph

3.1. Atomic Decomposition

For any directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the computation of all strongly connected components can be done in linear time wrt. the size of $\mathcal{G}$ which is defined as $\mathsf{size}(\mathcal{G}) = |\mathcal{V}| + |\mathcal{E}|$ [10].[10] In the case of directed hypergraphs, $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, the computation of all strongly connected components can be done in quadratic time wrt. the size of $\mathcal{H}$, where $\mathsf{size}(\mathcal{H}) = \sum_{(T,H)\in\mathcal{E}}(|T| + |H|)$ [1]. For a directed hyperedge (T, H) in $\mathcal{E}$, T is the set of tail nodes and H is the set of head nodes of the hyperedge.

In [7], we noticed that the axiom dependency hypergraph of some ontologies from the NCBO Bioportal, like CHEBI and Gazetteer, is a directed graph because all hyperedges are simple. That is, we can use an algorithm like Tarjan [10] to compute in linear time all the strongly connected components of the respective axiom dependency hypergraph.

For ontologies like Snomed CT and Full Galen, the respective axiom dependency hypergraph contains

[10] $|\mathcal{S}|$ denotes the cardinality of a set $\mathcal{S}$.

both, simple and complex hyperedges. For such hypergraphs, it is in general not possible to compute all strongly connected components in linear time. To speed up the computation of strongly connected components, we try to reduce the size of the original axiom dependency hypergraph. First, we compute the strongly connected components of the graph fragment of the axiom dependency hypergraph. This step can be completed in linear time wrt. the size of the graph fragment using, e.g., Tarjan's algorithm. Second, we collapse the strongly connected components of the graph fragment into single nodes. This implies that the set of nodes of each strongly connected component is replaced by a new node. Therefore, each edge that has one (or two) of the nodes that have been replaced by the new node will be re-computed to accommodate this new node (or disappear if the new tail and head nodes are the same). Notice that in HyS, an axiom dependency hypergraph is represented using a directed labelled graph. Thus, the labels of the new node must be computed. The set of MLSs of the new node includes all the different MLSs of the nodes that have been replace. Similarly, the signature symbols of the new node will include all the signature symbols of the nodes that have been replaced. As a result, we produce the partially condensed axiom dependency hypergraph (pcADH). Finally, in step three, we compute the strongly connected components of the pcADH by determining for any two nodes whether they are mutually connected. That is, we verify if there is a path from one node to the other and vice versa in the hypergraph. This last step produces the condensed axiom dependency hypergraph. Note that computing mutual reachability this way is a quadratic process wrt. the size of the pcADH [1]. However, it is usually more efficient as the number of nodes of the original ADH is typically reduced [7].

Figure 4 provides a more succinct description of the previous process.

function compute_condensed_hypergraph($\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$)
1: $\mathcal{G}_{pc}$:= collapse_SCCs($\mathcal{G}$, Tarjan($(\mathcal{V}, \mathcal{E})$))
2: if (contains_complex_Dependencies($\mathcal{G}_{pc}$) = false)
3: then return $\mathcal{G}_{pc}$
4: end-if
5: $\mathcal{G}_c$:= collapse_SCCs($\mathcal{G}_{pc}$, mutual_reach($\mathcal{G}_{pc}$))
6: return $\mathcal{G}_c$

Fig. 4. Atomic Decomposition Algorithm [7]

The function compute_condensed_hypergraph(.) gets as input a directed labelled graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$, that represents an axiom dependency hypergraph. In Line 1, the function collapse_SCCs(.,.) calls the function Tarjan(.) to identify the strongly connected components of the graph fragment of $\mathcal{G}$. The graph fragment of $\mathcal{G}$ corresponds with all the nodes of $\mathcal{G}$ ($\mathcal{V}$) but only simple directed hyperedges ($\mathcal{E}$). After the strongly connected components of the graph fragment of $\mathcal{G}$ are identified, the function collapse_SCCs(.,.) replaces the nodes of each strongly connected component by one new node. This implies that the simple directed edges that contain one or more of the replaced nodes are re-calculated to accommodate the new nodes. Notice that if the head and the tail nodes of a simple directed hyperedge are part of the same strongly connected component then this hyperedge is deleted from $\mathcal{E}$. Moreover, the labels of each new node has to be computed as it was explained earlier in this section. As a result, $\mathcal{G}_{pc}$ stores the directed labelled graph representation of the partially condensed axiom dependency hypergraph.

In Line 2, compute_condensed_hypergraph(.) checks if $\mathcal{G}$ has complex directed hyperedges. If it is not the case then the partially condensed axiom dependency hypergraph is also the condensed axiom dependency hypergraph and the function returns $\mathcal{G}_{pc}$. However, if there are complex directed hyperedges then the function collapse_SCCs(.,.) is executed again in Line 5. To compute the strongly connected components in $\mathcal{G}_{pc}$, mutual_reach(.) is called instead of Tarjan(.). As in Line 2, the nodes of each strongly connected components are replaced by new nodes. Like before, the function collapse_SCCs(.,.) re-calculates the simple directed hyperedges and computes the labels of the new nodes. As a result, the condensed axiom dependency hypergraph is stored in $\mathcal{G}_c$ which is returned by the function compute_condensed_hypergraph(.).

Let $\mathcal{O} = \{\alpha_1, ..., \alpha_8\}$ be an ontology, where each axiom is defined as follows:

- $\alpha_1 = A \sqsubseteq B$
- $\alpha_2 = B \sqsubseteq C$
- $\alpha_3 = C \sqsubseteq A \sqcap D \sqcap H$
- $\alpha_4 = D \sqsubseteq E \sqcap I$
- $\alpha_5 = E \sqsubseteq F$
- $\alpha_6 = F \sqsubseteq D$
- $\alpha_7 = G \sqsubseteq A \sqcap F$
- $\alpha_8 = H \sqcap I \sqsubseteq G$

Using the notion of $\bot$-locality, we build an axiom dependency hypergraph $\mathcal{H}^{\bot}_{\mathcal{O}} = (\mathcal{V}, \mathcal{E})$, where

- $\mathcal{V} = \mathcal{O}$
- $\mathcal{E} = \{e_1 = (\{\alpha_1\}, \alpha_2),\ e_2 = (\{\alpha_2\}, \alpha_3),$
  $e_3 = (\{\alpha_3\}, \alpha_1),\ e_4 = (\{\alpha_3\}, \alpha_4),$
  $e_5 = (\{\alpha_4\}, \alpha_5),\ e_6 = (\{\alpha_5\}, \alpha_6),$
  $e_7 = (\{\alpha_6\}, \alpha_4),\ e_8 = (\{\alpha_7\}, \alpha_1),$
  $e_9 = (\{\alpha_7\}, \alpha_6),\ e_{10} = (\{\alpha_8\}, \alpha_7),$
  $e_{11} = (\{\alpha_3, \alpha_4\}, \alpha_8)\}$

Figure 5 shows the strongly connected components of $\mathcal{H}^{\perp}_{\mathcal{O}}$ when only simple directed hyperedges are considered. These strongly connected components are computed using the function Tarjan(.). The function collapse_SCCs(.,.) collapses the strongly connected components into single nodes and produces the partially condensed version of the $\mathcal{H}^{\perp}_{\mathcal{O}}$. This can be seen in Figure 6. There is a complex directed hyperedges, e_{11}. Then, the function collapse_SCCs(.,.) is called again and the function mutual_reach(.) is executed to compute the remained strongly connected components. The strongly connected component detected is collapsed into a single node. Then, the function compute_condensed_hypergraph(.) returns the condensed version of $\mathcal{H}^{\perp}_{\mathcal{O}}$ that it is shown in Figure 7.

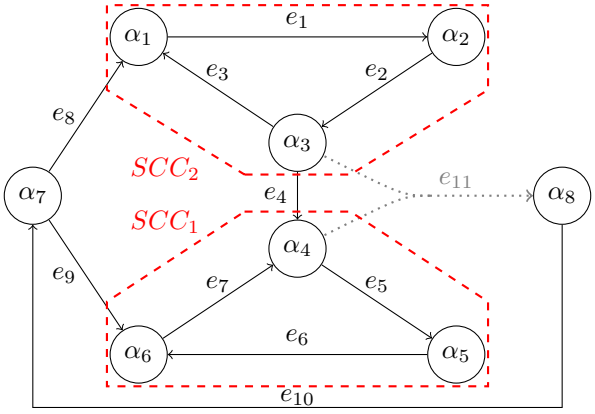

Fig. 5. SCCs in the graph fragment of the ADH

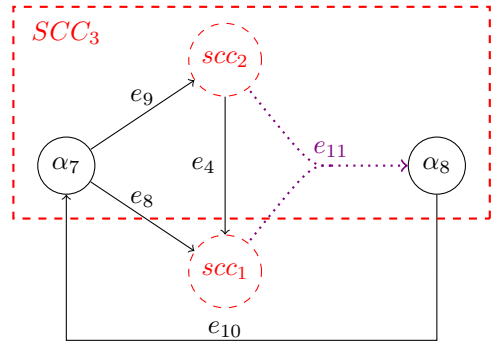

Fig. 6. SCCs in the partially condensed ADH

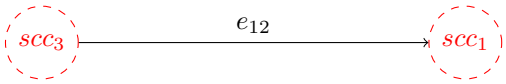

Fig. 7. Condensed ADH

3.2. Module Extractor

Locality-based modules correspond to connected components in the axiom dependency hypergraph or its (partially) condensed version [7]. Therefore, it is possible to compute a locality-based module of an ontology for a given signature by simply computing the connected component in the directed labelled graph representation of an axiom dependency hypergraph. In this section, we now present an algorithm that receives as an input a directed labelled graph and a signature; it computes a connected component; and it returns the locality-based module that corresponds to the connected component that was calculated.

To compute a connected component in a directed graph (or hypergraph), we need one or more initial nodes that are used to identify which other nodes are connected from this initial set of nodes. The union of the set of initial nodes and the connected nodes identified forms a connected component. For a given input signature Σ, a node of the directed labelled graph is an initial node if at least one of its minimal non-locality signatures is in Σ. Similar to the module extraction algorithm described in Section 2.1, each time that a new connected node is found, the input signature Σ is extended with the signature of the axiom (or axioms) represented by this node.

To identify new connected nodes in a directed labelled graph representation of an axiom dependency hypergraph, there are two main complementary approaches. The first approach takes advantage of the fact that all simple directed hyperedges are explicitly represented in the directed labelled graph. In HyS, the simple directed hyperedges are represented using an adjacency list data-structure that provides efficient access to direct successors of a given node. Therefore, the algorithm first searches for successors connected via simple directed hyperedges from the nodes that have been already identified as part of the connected component that it is being computed. The second approach explores complex directed hyperedges that are encoded in the labels of each node of the directed labelled graph. To achieve this, the algorithm searches for nodes that are not part of the current version of the

connected component and they have at least one minimal non-locality signature that is in Σ (which has been extended with the signatures of the nodes already included in the connect component).

When no new nodes can be added into the connected component, the algorithm identifies which are the axioms that correspond with the nodes in the connected component. These axioms represent the locality-based module for Σ and the ontology from which the directed labelled graph was built.

Figure 8 provides a more concrete description of the algorithm for computing locality-based modules given a directed labelled graph and a signature.

---

function $\mathsf{Mod}^x(\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}), \Sigma)$

1: $\Sigma_0 := \Sigma$, $m := 1$

2: $\mathcal{S}_0 := \emptyset$, $\mathcal{S}_1 := \{v \in \mathcal{V} \mid \Sigma_v \subseteq \Sigma_0$ for some $\Sigma_v \in \mathsf{MLS}^x(v)\}$

3: do

4: $\quad m := m + 1$

5: $\quad \mathcal{S}_m := \bigcup \{\mathcal{E}(v) \mid v \in \mathcal{S}_{m-1} \setminus \mathcal{S}_{m-2}\} \cup \mathcal{S}_{m-1}$

6: $\quad \Sigma_m := (\bigcup_{s \in \mathcal{S}_m \setminus \mathcal{S}_{m-1}} \mathsf{sig}(s)) \cup \Sigma_{m-1}$

7: $\quad \mathcal{S}_m := \mathcal{S}_m \cup \{v \in \mathcal{V} \mid \Sigma_v \subseteq \Sigma_m$ for some $\Sigma_v \in \mathsf{MLS}^x(v)$ with $|\Sigma_v| > 1\}$

8: until $\mathcal{S}_m = \mathcal{S}_{m-1}$

9: return $\mathsf{get\_axioms}(\mathcal{S}_m)$

Fig. 8. Module extraction algorithm [7]

In Line 2, the algorithm determines the set $\mathcal{S}_1$ of initial nodes in $\mathcal{G}$. Every initial node $\mathcal{S}_1$ is associated with a minimal non-locality signature that is contained in Σ. In Line 5, the set of nodes is determined that are reachable via simple directed hyperedges that are explicitly given in $\mathcal{E}$. Note that $\mathcal{E}(v)$ denotes the set of nodes that are directly reachable in $\mathcal{G}$ from the node v using simple directed hyperedges. In Line 7, the input signature is extended with the symbols that are associated to the nodes reached so far. Using the extended signature Σ_m, the function $\mathsf{Mod}^x(.,.)$ computes the nodes that can be reached using complex directed hyperedges implicitly represented by the labels $\mathcal{L}(v)$ of the nodes v in $\mathcal{S}_m$. The algorithm iterates until a fixed point is reached and no more new nodes are added.

4. Interfaces of HyS

The current version of HyS provides two main interfaces: an API and a command line interface. The API simplifies the access to the main features of HyS from other applications. The command line interface allows users to indicate which operation they would like to execute, where is located the input data and where should be store the results of the execution of HyS.

4.1. Application Programming Interface

The Java class '*HyS*' provides the main methods to compute a (partially) condensed axiom dependency hypergraph and extract atoms, its dependencies and syntactic local modules. Given an input ontology, the class HyS provides access to the following features:

- compute the (partially) condensed axiom dependency hypergraph,
- retrieve the axioms that correspond to a given node,
- retrieve the child (parent) nodes of a given node,
- retrieve the set of nodes that do not have incoming hyperedges (root nodes),
- retrieve the set of nodes that do not have outgoing hyperedges (leave nodes),
- retrieve the labels of a given node, and
- retrieve the syntactic locality-based module for a given signature node.

The child (parent) nodes of a node v are the nodes that are in the head (tail) of simple directed hyperedges such that v is the tail (head) of these hyperedges. Moreover, the labels of a node v are the minimal non-locality signatures and the signatures of the axiom(or axioms) that are represented by v.

The API of HyS is built on top of the OWLAPI. Therefore, it is possible to use the classes and methods provided by the OWLAPI to store set of axioms as OWL ontologies.

Figure 4 shows an example on how to create a condensed axiom dependency hypergraph; how we can retrieve the axioms that are represented by each node in the condensed axiom dependency hypergraph; and how we can compute a connected component for a given signature that corresponds to a locality-based module for that signature.

In Line 1, we create a HyS object that computes an axiom dependency hypergraph taking an ontology and a syntactic locality notion as an input. We use the algorithm by Tarjan to determine the strongly connected components of the graph fragment of the axiom dependency hypergraph. The method `condense(.)` will collapse the strongly connected components into single nodes to produce the partially condensed axiom

```
void Example(OWLOntology o, Set<OWLEntity> s){
1: HyS h = new HyS(o, ModuleType.BOT);
2: h.condense(SCCAlgorithm.TARJAN);
3: h.condense(SCCAlgorithm.MREACHABILITY);
4: Set<Node> allNodes = h.getNodes();
5: Iterator nItr = allNodes.iterator();
6: while (nItr.hasNext()) {
7:   Set<OWLAxiom> atom =
            h.getAxioms(nItr.next());
8: }
9: Set<Node> connectedComponent =
            h.getConnectedComponent(s);
10:Set<OWLAxiom> module =
            h.getAxioms(connectedComponent);
}
```

Fig. 9. Example of the HyS API

dependency hypergraph (cf. Line 2). Then we use the mutual reachability algorithm to compute the strongly connected components of the partially condensed axiom dependency hypergraph. In Line 3, we apply the method `condense(.)` again to collapse the strongly connected components to produce the condensed axiom dependency hypergraph. Lines 4 to 8 illustrate how to compute the atoms of the ontology by first collecting the nodes of the condensed axiom dependency hypergraph and subsequently retrieving the axioms represented by each node. In Line 9, we show how to compute a connected component given a signature. As for atoms, it is possible to retrieve the axioms represented by the nodes of the connected component, which corresponds to the syntactic locality based module of the ontology for the signature (cf. Line 10).

4.2. *Command Line Interface*

It is also possible to execute HyS as a standalone application. Using the command line interface, we can ask HyS to compute and store the atomic decomposition of an ontology or the syntactic locality-based modules for a collection of input signatures. Figure 10 shows the possible parameters for HyS.

For computing the atomic decomposition of an ontology, we have to indicate the location of the input ontology and the folder for storing the atoms together with their dependencies.

For computing syntactic locality-based modules, we have to provide the location of the input ontology, the folder where the input signatures are stored. The input signature of each module that we want to compute using HyS must be stored in a text file with extension '.sig' that contains one signature symbol per line. The computed modules will be stored as OWL-ontologies in the same folder that contains the signatures. It is also possible to indicate to HyS to compute the modules using the axiom dependency hypergraph or its (partially) condensed version.

```
java -jar hys.jar -A -o <path-to-ontology>
                     -d <path-to-folder>
java -jar hys.jar -M <-da|-dp|-dc>
                     -o <path-to-ontology>
                     -d <path-to-folder>
options:
 -A  stores the atomic decomposition
 -M  stores syntactic locality-based modules
 -da computes only the ADH
 -dp computes only the pcADH
 -dc computes the cADH
 -o  input ontology
 -d  folder for reading signatures and saving
     AD/modules
```

Fig. 10. Commands available in HyS

5. Architecture of HyS

The architecture of HyS consists of three main components: *Ontology Manager*, *Graph Builder* and *Strong Connectivity Library*. Extensibility was one of the main requirements that we have in mind when we designed the architecture of HyS. In particular, we are interested in incorporating new algorithms that improve the performance of HyS for computing the strongly connected components of directed hypergraphs. This is why the algorithms are organized in a library that can easily be extended with new algorithms. Next, we provide a description of each of the main components.

5.1. *Ontology Manager*

The *Ontology Manager* has been implemented on top of the OWLAPI and it provides the required functionality to load OWL ontologies and store the atomic decomposition of an ontology and syntactic locality-based modules. The Ontology Manager provides methods to *normalise* and *denormalise* axioms. An OWL axiom that is not normalised may have

exponentially many minimal non-locality signatures, whereas a normalised axiom has at most two such signatures [7]. To normalise an axiom (or an ontology) formulated in OWL 2, we can apply the normalisation rules presented in [8]. Denormalisation is the reverse process required to retrieve the original axioms (ontology) from their normalised versions. Both, normalisation and denormalisation, can be executed in linear time wrt. the size of the input ontology [8]. For OWL-EL ontologies, normalisation and denormalisation are simple processes that can be executed very efficiently. These are the processes that have been implemented in the current version of HyS. We plan to extend the implementation of the Ontology Manager component to normalise and denormalise OWL 2 ontologies.

5.2. Graph Builder

The Graph Builder is the core component in the HyS architecture as it interacts with all other components. The essential task is to build the directed labelled graph and all related data-structures defined to speed up the computation of the algorithms for computing connected and strongly connected components. The Graph Builder also provides a method for collapsing a set of nodes into a single node which is required to compute the (partially) condensed version of an axiom dependency hypergraph. The nodes of the directed labelled graph are identified internally using integers. The simple directed hyperedges are represented in an adjacency list data structure containing pairs of nodes. Symbols of the input ontology are also represented internally using integers. We have implemented specific data-structures that can facilitate the execution of subset and superset queries over sets of symbols which are frequent operations during the computation of connected and strongly connected components.

The manipulation of the data structures defined by the Graph Builder can only be done using specific methods. The idea is to provide an internal interface between the Graph Builder and the algorithms defined in the Strong Connectivity library. Using this approach, the algorithms for computing strongly connected components will not be affected by changes in the implementation of the Graph Builder and it will facilitate the extension of the Strong Connectivity library with new algorithms.

The selection of $\bot$-locality or $\top$-locality will only affect the computation of minimal non-locality signatures. The algorithm for computing connected components (which correspond to syntactic locality-based modules) is independent of the locality notion used to build the directed labelled graph. Thus, a dedicated library of algorithms for computing connected components is not needed.

5.3. Strong Connectivity Library

This component provides a collection of algorithms to compute the strongly connected components of a directed hypergraph (represented as a directed labelled graph). The algorithms do not directly manipulate the internal data-structures that the Graph Builder has created. Instead, they use the methods provided by the Graph Builder to access to the data structures.

In the current version of HyS, the Strong Connectivity Library includes two algorithms: Tarjan for directed graphs and mutual reachability for directed labelled graphs. The implementation of Tarjan's algorithm ignores the labels of a directed labelled graph. It only uses the simple directed hyperedges stored in the adjacency list data structure created by the Graph Builder. Therefore, Tarjan's algorithm is only used to compute the strongly connected components on the graph fragment of the axiom dependency hypergraph. This is done to compute the partially condensed axiom depedency hypergraph.

The second algorithm, mutual reachability, takes into account the labels of the directed labelled graph. The labels encode all the directed hyperedges of an axiom dependency hypergraph. Therefore, the implementation of mutual reachability is used to compute the strongly connected components in the partially condensed axiom dependency hypergraph. This step is required to compute the condensed axiom dependency hypergraph from which is possible to calculate the atomic decomposition of an ontology.

6. Evaluation of HyS

For the evaluation of HyS v1.0, we have selected nine well-known biomedical ontologies. Seven of them are available in the NCBO Bioportal.[11] The version of Full-Galen that we used is available in the Oxford ontology repository.[12]

We divide the ontologies into two groups: a group consisting of CHEBI, FMA-lite, Gazetteer, GO, NCBI and RH-Mesh, and another group consisting of CPO,

[11] http://bioportal.bioontology.org/
[12] http://www.cs.ox.ac.uk/isg/ontologies/

Full-Galen and SNOMED CT. The axiom dependency hypergraph of each ontology of the former group can be represented using a directed graph. This means that the axiom dependency hypergraph does not have complex directed edges. In the case of the ontologies of the latter group, the axiom dependency hypergraph also contains complex directed hyperedges

For testing atomic decomposition, we compare HyS against two systems: FaCT++ v1.6.3 and OWLAPI-AD v4.0.1. They represent the state of the art applications for computing the atomic decomposition of OWL 2 ontologies and they implement the same algorithms. This is interesting to measure the impact of using two different programming languages (C++ and Java).

For testing module extraction using syntactic locality, we compare HyS against FaCT++ v1.6.3 and OWLAPI v3.5.0 (module extraction using the latest version of the OWLAPI, v4.0.1, failed for some ontologies). OWLAPI is a widely used tool for computing syntactic locality-based modules. We choose OWLAPI instead of OWLAPI-AD because we would like to verify how much FaCT++ has improved the execution of the OWLAPI when it computes the same syntactic locality-based modules. We decide to do not include CEL in the evaluation. The last version of CEL was released in 2009 and it seems that the development and maintenance of this tool stopped some years ago.

All experiments were conducted on an Intel Xeon E5-2640 2.50GHz with 100GB RAM running Debian GNU/Linux 7.6 and Java 1.8.0_25.

Table 1 lists the time needed for each system to compute the atomic decomposition of the ontologies. The time values are the average of at least 10 executions. We applied a timeout of 48h, which aborted the execution of OWLAPI-AD on the ontology Gazetteer. OWLAPI-AD also could not deal with FMA-lite due to out-of-memory errors. Moreover, the table contains, for each ontology, the size of the signature, the number of axioms of the form $A \sqsubseteq C$, where A is a concept name, the number of axioms of the form $C \equiv D$, the number of role axioms contained in the ontology.

HyS consistently outperforms FaCT++ which in turn (considerably) outperforms the OWLAPI-AD. In the case of the first group of six ontologies, an over 1 000-fold speedup could be achieved compared to the performance of FaCT++ on FMA-lite and Gazetteer. For the smallest ontology in this group, which is GO, HyS is 11 times faster than FaCT++. HyS also scales better than the other systems. For the second group of three ontologies, the speedup is reduced but HyS is still considerably faster. HyS is 4–6 times faster than FaCT++ and 8–25 faster than the OWLAPI-AD.

Table 2 lists, for each ontology, the number of axioms whose minimal locality signature contains more than one symbol. Such axioms may require complex hyperedges to represent the $\bot$-locality dependencies to other axioms. This is why the second group of ontologies cannot be represented using a directed graph like in the first group. Table 2 also shows a significant reduction of the number of nodes and directed edges after the axiom dependency graph is partially condensed. This reduction of nodes and directed edges contributes to decrease the time needed to compute all the strongly connected components of the axiom dependency hypergraph.

We also compare the performance of HyS for extracting $\bot$-locality modules with the performance of FaCT++ v1.6.3 and the OWLAPI v3.5.0. Table 3 presents, for every system, the time needed to extract a module from an ontology for a signature consisting of 500 symbols selected at random. Every time value in the table is the average taken over 1000 executions.

HyS outperforms FaCT++ and the OWLAPI in all cases. For the first group of six ontologies, the best speedup of over 90 times was achieved in the case of FMA-lite. Notice that module extraction times using the pcADH and the cADH (last two columns) are nearly the same as the two graphs are equivalent. The small variation in extraction time is due to noise in the execution environment. The differences in the times values in the third column and the last two columns correspond to the differences in size of the ADH and the pcADH/cADH. For the second group of three ontologies, the best performance improvement was realised in the case of Full-Galen with a speedup of over 20-times. However, we note that using the cADH instead of the pcADH does not yield a large performance difference despite the fact that the cADH is slightly smaller than the pcADH. In the particular case of Full-Galen, there appears to be a trade-off between condensation and increased time needed to perform signature containment checks. Computing the partially condensed ADH (using a linear time algorithm) is generally much faster than computing the condensed ADH (which is done in quadratic time). Given the that module extraction times are similar when using the pcADH and the cADH (cf. the times in the last two columns), it seems more efficient to only compute modules using the partially condensed ADH.

Table 1
Evaluation of HyS for Atomic Decomposition.

Ontology $\mathcal{O}$	Properties of $\mathcal{O}$				Time for Atomic Decomposition of $\mathcal{O}$		
	Signature size	#axioms $A \sqsubseteq C$	#axioms $C \equiv D$	#role axioms	FaCT++ v1.6.3	OWLAPI-AD v4.0.1	HyS v1.0
CHEBI	37 891	85 342	0	5	127 s	1 904 s	4 s
FMA-lite	75 168	119 558	0	3	15 861 s	–	16 s
Gazetteer	517 039	652 355	0	6	26 957 s	–	24 s
GO	36 945	72 667	0	2	43 s	1 739 s	4 s
NCBI	847 796	847 755	0	0	41 854 s	71 121 s	65 s
RH-Mesh	286 382	403 210	0	0	5 844 s	11 395 s	17 s
CPO	136 090	306 111	73 461	96	8 345 s	31 247 s	2 269 s
Full-Galen	24 088	25 563	9 968	2 165	564 s	957 s	114 s
SNOMED CT	291 207	227 698	63 446	12	13 969 s	63 823 s	2 523 s

Table 2
Size of the axiom dependency hypergraphs.

Ontology $\mathcal{O}$	#axioms α with $\lvert\mathsf{MLS}(\alpha)\rvert > 1$	Statistics of the axiom dependency hypergraph for $\mathcal{O}$					
		ADH		pcADH		cADH	
		#nodes	#edges	#nodes	#edges	#nodes	#edges
CHEBI	0	85 344	341 079	32 913	74 204	32 913	74 204
FMA-lite	0	165 786	991 137	100 873	56 948	100 873	56 948
Gazetteer	0	652 361	3 703 218	517 028	1 089 717	517 028	1 089 717
GO	0	273 082	202 175	245 259	69 023	245 259	69 023
NCBI	0	847 755	847 719	847 755	847 719	847 755	847 719
RH-Mesh	0	403 210	1 032 421	286 263	401 390	286 263	401 390
CPO	73 464	379 734	3 725 565	131 595	779 228	129 157	764 956
Full-Galen	9 968	37 696	323 836	20 396	58 759	15 211	38 746
SNOMED CT	59 183	573 255	2 494 041	304 770	913 347	304 523	911 581

Table 3
Evaluation of HyS for module extraction.

Ontology $\mathcal{O}$	Time for Extraction of $\bot$-local modules from $\mathcal{O}$				
	FaCT++ v1.6.3	OWLAPI v3.5.0	HyS v1.0 ADH	HyS v1.0 pcADH	HyS v1.0 cADH
CHEBI	39.3 ms	175.3 ms	3.8 ms	2.3 ms	2.1 ms
FMA-lite	329.2 ms	1 041.7 ms	55.4 ms	3.7 ms	3.2 ms
Gazetteer	181.3 ms	1 502.4 ms	27.2 ms	15.9 ms	15.7 ms
GO	519.5 ms	1 395.8 ms	8.2 ms	6.2 ms	5.9 ms
NCBI	238.7 ms	9 189.2 ms	22.5 ms	15.8 ms	15.9 ms
RH-Mesh	93.0 ms	1 810.7 ms	10.4 ms	9.1 ms	8.8 ms
CPO	574.9 ms	3 024.2 ms	84.3 ms	53.3 ms	51.4 ms
Full-Galen	76.2 ms	214.8 ms	13.1 ms	3.7 ms	2.9 ms
SNOMED CT	534.3 ms	2 839.9 ms	92.9 ms	87.2 ms	83.9 ms

7. Conclusions and Future Work

We have presented the tool HyS, a Java implementation for computing the atomic decomposition and locality-based modules of OWL-EL ontologies. HyS outperforms FaCT++ and the OWLAPI-AD in computing the atomic decomposition of all biomedical ontologies tested. In some cases a staggering speedup of over $1\,000$ times could be realised. Moreover, HyS significantly outperforms FaCT++ and the OWLAPI in extracting syntactic $\bot$-locality modules.

We plan to extend HyS to support both $\top$-locality and full $\mathcal{SROIQ}$-ontologies. Moreover, it would be interesting to investigate the possibility to compute strongly connected components in hypergraphs in less than quadratic time. Such a result would improve the performance of computing mutual reachability in the axiom dependency hypergraph for ontologies whose locality-based dependencies can only be represented by hyperedges with more than one node in the tail.

References

[1] X. Allamigeon. Strongly connected components of directed hypergraphs. *CoRR*, abs/1112.1444, 2011.

[2] F. Baader, S. Brandt, and C. Lutz. Pushing the el envelope further. In K. Clark and P. F. Patel-Schneider, editors, *In Proceedings of the OWLED 2008 DC Workshop on OWL: Experiences and Directions*, 2008.

[3] B. Cuenca-Grau, I. Horrocks, Y. Kazakov, and U. Sattler. Modular reuse of ontologies: theory and practice. *JAIR*, 31:273–318, 2008.

[4] C. Del Vescovo, D. D. G. Gessler, P. Klinov, B. Parsia, U. Sattler, T. Schneider, and A. Winget. Decomposition and modular structure of bioportal ontologies. In *Proc. of ISWC'11*, pages 130–145. Springer-Verlag, 2011.

[5] C. Del Vescovo, B. Parsia, U. Sattler, and T. Schneider. The modular structure of an ontology: Atomic decomposition. In *Proc. of IJCAI'11*, pages 2232–2237, 2011.

[6] F. Martín-Recuerda and D. Walther. Towards fast atomic decomposition using axiom dependency hypergraphs. In *Workshop on Modular Ontologies (WoMO) 2013*, CEUR Workshop Proceedings, pages 61–72. CEUR-WS.org, 2013.

[7] F. Martín-Recuerda and D. Walther. Fast modularisation and atomic decomposition of ontologies using axiom dependency hypergraphs. In *Proceedings of ISWC'14: the 13th International Semantic Web Conference*, volume 8797 of *Lecture Notes in Computer Science*, pages 49–64. Springer International Publishing, 2014.

[8] R. Nortje, A. Britz, and T. Meyer. Reachability modules for the description logic SRIQ. In *Proc. of LPAR-13*, 2013.

[9] B. Suntisrivaraporn. *Polynomial time reasoning support for design and maintenance of large-scale biomedical ontologies*. PhD thesis, TU Dresden, Germany, 2009.

[10] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[11] D. Tsarkov. Improved algorithms for module extraction and atomic decomposition. In *Proc. of DL'12*, volume 846 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[12] D. Tsarkov, C. Del Vescovo, and I. Palmisano. Instrumenting atomic decomposition: Software APIs for OWL. In *Proceedings of OWLED'13: the 10th International Workshop on OWL: Experiences and Directions*, volume 1080 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.