Divide et Impera: Metareasoning for Large Ontologies

Dmitry Tsarkov and Ignazio Palmisano

University of Manchester,
School of Computer Science,
Manchester, UK
{tsarkov, palmisai}@cs.man.ac.uk

Abstract. Ontologies have the nasty habit of growing in size and expressivity until all reasoners are at a loss to treat them in reasonable time. While it is widely known that the worst case complexity for OWL 2 DL reasoning is double exponential in time, but in fact most ontologies tend to be well behaved in practice, it is less well known that any ontology, left to itself, will grow until the worst OWL constructs gang up together and stop any reasoner from providing answers fast.
We present a brand new idea to tackle this issue, by using a metareasoner which leverages best of breed reasoners and modularisation techniques to prevent the worst side effects of growing ontologies and keep query answering performance to its best.
In our approach, each query will be answered by using only part of the ontology, and the best reasoner for the query will be selected on the basis of the features of this ontology portion, which is built using modularisation and/or atomic decomposition.
We show the performances of an implementation of this approach. Our Chainsaw metareasoner is not designed to deal with small and inexpressive ontologies, for which a regular reasoner is more than adequate. It, instead, deals with large, complex and unwieldy ontologies, for which better solutions are not (yet) available.

1 Introduction

As it is well known to ontology engineers, ontologies become harder to manage, in terms of understanding them and reasoning with them, in a way which is, unfortunately, not proportional to the increase of their size. It is often the case that a small number of changes makes an ontology much harder for a reasoner to process. However, keeping ontologies small and simple is not always possible, because they might fail to satisfy the application requirements. On the other hand, it is possible for a small ontology to be hard to reason about.

How to best divide an ontology into smaller portions according to the needs of the task at hand is still an open problem; the rule of thumb that many approaches follow is that, when in doubt, one should try to keep the ontology as small as possible.

This is the main intuitive reason for modularization, i.e., a set of techniques for splitting ontologies into fragments without loss of entailments for a given set of terms. To date, however, not many tools provide either user support for these techniques, nor leverage them for reasoning tasks.

Chainsaw, our metareasoner implementation prototype, exploits the idea in the following way. For every inference query it creates a module of the ontology that is then used to answer the query. The module is created in a way that guarantees the result of the query should be the same as for the whole ontology, i.e., there is no loss of entailments.

Chainsaw is designed as a wrapper for other reasoners. It uses reasoner factories to build reasoners on modules of its root ontology, and will integrate the ability to choose the reasoner best suited to each reasoning task on the basis of the module characteristics, such as size and/or expressivity. The most obvious characteristic is an OWL 2 profile of the module. E.g., if the profile is OWL 2 EL then some efficient EL reasoner, like ELK [4], could be used to achieve the best performance.

The advantages of using modules instead of whole ontologies inside a reasoner reside in the simplification of reasoning. The complexity of reasoning in OWL 2 DL is N2EXPTIME-hard on the size of the input, therefore being able to divide the ontology is likely to produce performance improvements orders of magnitude larger than the relative reduction in size.

Moreover, using modules inside the reasoner can help the knowledge engineer to concentrate on modelling the knowledge instead of worrying about the language pitfalls, since the extra complexity can be tackled in a transparent way. This, however, does not mean that complexity is no longer an issue. Modularisation is not a silver bullet for reasoning, as in some cases the module needed to answer a query might still include most of the ontology.

Another advantage of Chainsaw architecture is that it is able to use different OWL reasoners for each query and module; this allows for choosing the reasoner best suited for the OWL 2 profile of a specific module. In those cases where it is not clear which reasoner is best, it is possible to start more than one reasoner and simply wait for the first one to finish the task - this might require more resources, but statistical records can be kept to improve future choices. This functionality, however, is not yet implemented.

The reverse of the medal is that, for simple ontologies, Chainsaw is likely to be slower than most of the reasoners it uses; this derives from the overhead needed to manage multiple reasoners and modularization of the ontology itself. Our objective, in this paper, is to illustrate an approach that can squeeze some answers out of ontologies that are too troublesome for a single traditional reasoner to handle.

In Section 2, we describe briefly the theory behind Atomic Decomposition and Modularization, which are the building blocks of this approach; in Section 3 we describe the implementation details, trade-offs and strategies adopted, and in Section 4 we present some preliminary results on the effectiveness of Chainsaw.

2 Atomic Decomposition and Modularization

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 $\widetilde{\alpha}$ a signature of that axiom, i.e. a set of all entities in α. The same notation is also used for the set of axioms.

Definition 1 (Query). *Let O be an ontology. An axiom α is called an* entailment *in O if $O \models \alpha$. A check whether an axiom α is an entailment in O is an* entailment query*. A* subclass (superclass, sameclass) query *for a class C in an ontology O is to determine a set of classes $D \in \widetilde{O}$ such that $O \models C \sqsubseteq D$ (resp. $O \models D \sqsubseteq C, O \models C = D$). A* hierarchical query *is either subclass, superclass, or sameclass query.*

Definition 2 (Module). *Let O be an ontology and Σ be a signature. A subset M of the ontology is called* module *of O w.r.t. Σ if for every axiom α such that $\widetilde{\alpha} \subseteq \Sigma$, $M \models \alpha \iff O \models \alpha$.*

One way to build modules is through the use of axiom *locality*. An axiom is called ($\bot$-) local w.r.t. a signature Σ if replacing all entities not in Σ with $\bot$ makes the axiom a tautology. This syntactic approximation of locality was proposed in [1] and provides a basis for most of the modern modularity algorithms [3].

This modularisation algorithm is used to create an atomic decomposition of an ontology, which can then be viewed as a compact representation of all the modules in it [7].

Definition 3 (Atomic Decomposition). *A set of axioms A is an* atom *of the 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 for every module M if $A \subseteq M$ then $B \subseteq M$. 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.*

The dependency closure of an atom, computed by following its dependencies, constitutes a module; this module can then be used to answer queries about the terms contained in this closure.

However, for a hierarchical query the signature would contain a single entity, but the answer set would contain entities that might not be in the module built for that signature.

In order to address this problem, we use Labelled Atomic Decomposition (LAD), as described in [6].

Definition 4 (Labelled Atomic Decomposition). *A* Labelled Atomic Decomposition *is a tuple $LAD = \langle S, \preccurlyeq, L \rangle$, where $G = \langle S, \preccurlyeq \rangle$ is an atomic decomposition and L is a labelling function that maps S into a set of labels. A* top-level labelling *maps an atom A to a (possibly empty) subset of its signature $L(A) = \widetilde{A} \setminus (\bigcup_{B \preccurlyeq A} L(B))$.*

Proposition 1. *Let $LAD = \langle S, \preccurlyeq, L\rangle$ be a top-level labelled atomic decomposition of an ontology O. Then for all named classes x, y from $\widetilde{O}$ the following holds:*

1. *If $O \models x \sqsubseteq y$, then $\exists A, B \in S : x \in L(A), y \in L(B)$ and $B \preccurlyeq A$;*
2. *If $O \models x \doteq y$, then $\exists A \in S : x \in L(A)$ and $y \in L(A)$.*

Proof. 1) From [3], Proposition 11, $O \models x \sqsubseteq y$ iff for the module M of O w.r.t. signature $\{x\}$ holds $M \models x \sqsubseteq y$. Assume $O \models x \sqsubseteq y$. Then M is non-empty and $x \in \widetilde{M}$. Thus there is an atom $A \in S$ such that $x \in L(A)$. Due to the atomic decomposition properties, the union of an atom together with all the dependent atoms forms a module. So let $M_A = \bigcup_{B \preccurlyeq A} B$ be such a module. This module also has x in its signature, so $M \subseteq M_A$. But by the definition of the top-level labelling M_A is the smallest module that contains x in the signature; so $M = M_A$. This also means that there is only one atom which label contains x. Now, using the results from [3], we can conclude that $y \in \widetilde{M_A}$; that means, that one of the atoms $B \in M_A$ is labelled with y. But all such atoms are dependencies of A, i.e. $B \preccurlyeq A$.

2) Assume $O \models x \doteq y$, which is equivalent to $O \models x \sqsubseteq y$ and $O \models x \sqsubseteq y$. From Case 1) this means that there are atoms A, A', B, B' such that $x \in L(A), x \in L(A'), y \in L(B), y \in L(B')$ and $B' \preccurlyeq A, A' \preccurlyeq B$. As shown in Case 1), there is only one atom that contains x (resp. y) in its label, so $A = A', B = B'$ and $B \preccurlyeq A, A \preccurlyeq B$. The latter is possible only in case $A = B$. □

This proposition provides a way to separate parts of the ontologies necessary to answer hierarchical queries about named classes. Indeed, it is enough to label the atomic decomposition with a top-level labelling and the modules for finding a subsumption relation could easily be found. This approach is orthogonal to a modularity-based one: while the latter deals easily with the entailment-like queries, the former provides a way to describe an ontology subset suitable to answer hierarchical queries.

3 Implementation of Chainsaw

The essence of Chainsaw is mirrored in the paper's title. Unlike other reasoners, which usually do the classification of the ontology before any query is asked, Chainsaw deals with requests in a lazy way, leaving classification to the delegate reasoners, which are usually at work on a small subset of the ontology.

For each query received, Chainsaw tries to keep the subset of the ontology needed to answer as small as possible without sacrificing completeness. This is achieved using different strategies according to the query; i.e., it is not possible to reduce the size of the ontology when checking for its consistency; however, other queries, as detailed in Section 2, can be answered by using modules built via LAD or locality based modules. More in detail, querying about superclasses of a term will only need the dependency closures of the top-level atoms for that term for the answers to be computed; the opposite is true for subclass requests.

During preprocessing of the ontology, a LAD of that ontology is built, using the Atomic Decomposition algorithm available in FaCT++ [5], and both dependency closure and its reverse are cached for every class name. For every query the module is constructed: via modularisation algorithm for entailment queries and via LAD for hierarchical queries. Then a suitable reasoner is created for that module, and the query is delegated to it. The answer then is returned to a user.

A naive strategy for answering any query would consist of:

- Build a module M for the query
- Start a new reasoner R on M
- Answer the original query using R

However, it is easy to find possible optimizations to this strategy.

First, this approach creates a new reasoner for each query; if two queries with the same signature are asked, two (identical) modules would be built and two reasoners would be initialized, while just keeping the same reasoner would be enough.

Moreover, while the number of possible signatures for a query is exponential in the size of the ontology signature (not counting possible fresh entities used in the query), the number of distinct modules that can be computed against a given ontology with these signatures is much smaller [2]. This means that, given a module, there is a good chance that it can be reused for answering queries with a slightly different signature; therefore, the same reasoner can be used to answer more than one specific query.

Therefore, a trade-off exists between reducing the size of the module to be reasoned upon, the complexity of determining such a module and the cost of starting a new reasoner for each query; to this, one must add the memory requirements of keeping a large module and reasoner cache.

Our approach in Chainsaw is to use a cache for modules and a cache for reasoners, both limited in the number of cached elements, and ordered as LRU caches; this has shown to perform rather well in some of our tests, reported in Section 4.1, where around one hundred thousand entailment checks against a large ontology have been satisfied using approximately 100 simultaneous reasoners, some of which were reused up to 20 times before being discarded. Similar results have been obtained when caching the modules to avoid rebuilding the same module for the same signature.

3.1 Future Improvements

There is one more optimization that was not implemented: if a module is included in another module, the larger module can be used in place of the smaller one. However, this presents a slippery slope problem: at what level do we stop using the next containing module, since we do not have an easy way to predict where this series of modules will become really hard to reason with?

Determining the containment is also an expensive operation; for simple modules, this operation might cost more than the actual reasoning required. The sweet spot for this optimization is a situation in which many fairly complex modules share a large number of axioms and are used often, and their union does not produce a module which pushes the reasoner's envelope. Using the union would provide for a good boost in performance and save memory as well, but at the time of writing we do not have an effective way of finding such spots.

It seems that atomic decomposition could provide relevant information for this task; an educated guess would be that such sweet spots reside near the parts of the dependency graph where a number of edges converge, but, to the best of our knowledge, there is no strong evidence in favor of this correlation. Future work might well explore this area.

Another improvement is to add a strategy to choose the best suited reasoner for a given module; such a strategy would have to take into account the known weak spots and strong points of each reasoner, as well as the characteristics of the module and of the query, such as size and OWL profile, or whether the query requires classification of the module or not. Where this is not sufficient, statistical records could be kept in order to create evidence based preferences and improve the strategy over time.

3.2 Integration in Existing Toolchains

Chainsaw implements the `OWLReasoner` interface defined in the OWL API[1]; it is therefore easy to add it to existing applications, such as ontology editors, which already use some of the existing implementations of `OWLReasoner`.

4 Empirical Evaluation

To check the performance of Chainsaw we ran several tests with it. For the tests we used a MacBooc Pro laptop with 2.66 GHz i7 processor and 8Gb of memory.

We use Chainsaw with FaCT++ v 1.5.3 as a delegate reasoner, and compare the results with the same version of FaCT++.

As a test suite we took 57 of the BioPortal ontologies[2]. For every ontology we perform a set of classes, properties and individuals tests, i.e., for every class we checked its satisfiability, ask for its instances and for sub-, super- and equivalent classes. Moreover, for every class we took its subclass C and superclass D, and ask whether C is a subclass of D. For every individual we ask the sets of its types, and for same and different individuals. For every property, its sub-, super, and equivalent properties were queried, together with property range and domain.

The test results are presented in a few tables below. All the tables have the same format: the name of the method called, total number of calls for all tests, and the total time spend in that method for Chainsaw and FaCT++, given in milliseconds.

[1] http://owlapi.sourceforge.net
[2] http://bioportal.bioontology.org

Method	Number of calls	Chainsaw	FaCT++
getUnsatisfiableClasses	57	0	3
init	57	23,442	1,490
isConsistent	57	2,532	0
precomputeInferences	57	1	28,411

Table 1. Methods called once per ontology.

Method	Number of calls	Chainsaw	FaCT++
getDifferentIndividuals	132	314	327
getSameIndividuals	132	28	8
getTypes	264	73	25
getInstances	173204	522,645	1,903

Table 2. Methods pertaining to individuals.

Table 1 contains operations that are done only once for every ontology. First every reasoner is initialised, then it is asked to precompute inferences (in our tests reasoners were asked to precompute a class hierarchy). In the next step the consistency of an ontology is checked. After that, we asked reasoners whether there were any unsatisfiable classes.

It is easy to see that the initialisation of Chainsaw reasoner took much more time than the one for FaCT++. That overhead comes from the necessity of creating LAD for each ontology. Interesting to see, however, how the time is distributed between the precomputeInferences and isConsistent steps. In the case of Chainsaw nothing is done during the precomputing step, and there is a need to do work for checking the ontology consistency. On the other hand, FaCT++ tests consistency during classification, so answering that query costs nothing.

The data on individuals is presented in the Table 2. There were only 3 ontologies with individuals in our test suite, and the time needed for individual-related operations is similar for both reasoners, with the exception of getInstances case, in which FaCT++ behaves much better than Chainsaw.

However, the discerning reader will have noticed that there is a number of ontologies not containing any individual; yet, the reasoners are taking some time, in the case of Chainsaw quite some time, to answer the query with an empty set. This points to possible optimisations for both Chainsaw and FaCT++ itself.

Class-related queries are presented in the Table 3. Note that the hierarchical questions are faster for Chainsaw; this is mainly because there is a simple way to check them, based on LAD. Satisfiability and entailment queries, however, took more time in Chainsaw than in FaCT++; here the overhead to produce modules and to create a reasoner starts play a negative role.

Method	Number of calls	Chainsaw	FaCT++
getEquivalentClasses	86602	2,012	1,509
getSubClasses	259863	7,433	10,007
getSuperClasses	259863	5,059	8,866
isSatisfiable	86602	309,067	1,349
isEntailed	4631065	661,707	89,570

Table 3. Methods pertaining to classes and axiom entailment.

Method	Number of calls	Chainsaw	FaCT++
getDisjointDataProperties	287	8	1
getEquivalentDataProperties	287	12	6
getDataPropertyDomains	574	265	456
getSubDataProperties	574	243	9
getSuperDataProperties	574	21	6
getDisjointObjectProperties	662	83	14
getEquivalentObjectProperties	662	86	14
getInverseObjectProperties	662	81	13
getObjectPropertyDomains	1324	3,514	1,176
getObjectPropertyRanges	1324	2,194	621
getSubObjectProperties	1324	36,657	32
getSuperObjectProperties	1324	176	27
getObjectPropertyValues	3907	7,085	6,645
getDataPropertyValues	3960	815	10

Table 4. Methods pertaining to properties.

Queries about properties are presented in the Table 4. Besides a few queries (like getSubObjectProperties), the two reasoners appear to behave similarly on that data, i.e. the difference, comparing with Tables 2 and 3, is much smaller.

4.1 Genesis of Chainsaw and OBI Prototype Results

The idea for Chainsaw was born of a concrete need: experiments needed to be run to prove a theory, and one of the ontologies chosen for the task was the latest version of the OBI ontology[3].

The task of *Knowledge Exploration* is to explore the structure of a model for a given class expression. In particular, the tableau algorithm proves the satisfiability of a class by providing a tree-shaped model of that class. The knowledge exploration allows one to access that model, i.e. to know the form of the model as well as the labels of the nodes and edges of the model's tree.

These experiments are based on an extension to `OWLReasoner` that was designed to allow a developer to gain information about the completion graph that a tableaux reasoner builds for classes and properties; this interface, available in

[3] obi-ontology.org/

the OWL API as `OWLKnowledgeExplorerReasoner`, is, at the time of writing, only implemented by FaCT++, so said experiments had to rely on it to be executed.

Unfortunately, the latest version of OBI is not easily managed by FaCT++; classifying the ontology takes, in fact, four hours and approximatively six gigabytes of memory to reach 97%. Our tests needed to query for the subsumption hierarchy of almost all classes declared and used in OBI, and to check whether a given class expression would subsume said classes; this amounted to little more than a hundred thousand subsumption tests.

Using the same idea we illustrated as the basis for Chainsaw, we were able to write code that would extract a module for the query, classify it and answer the query, using FaCT++; the maximum amount of time needed to classify one of these modules was five seconds, and the memory used for each classification was proportionally small; we were able to use caching of modules and reasoners effectively, and the whole experiment could be run in under four hours, using approximatively three gigabytes of RAM.

These (highly experimental) results are reported in Table 4.1. As is it easy to see, for this ontology Chainsaw is much better than FaCT++, as the latter could not provide *any* results in a given time frame. As we mentioned in Section 1, this is the kind of task for which Chainsaw was designed.

Reasoner	Classification time	Total memory	Subsumption tests	Time	Tests per second	Reasoner instances
FaCT++	> 4 hrs	6 GB	-	-	-	1
Chainsaw	-	3 GB	100,000	4 hrs	7	300

Table 5. Prototype performances on OBI ontology for knowledge exploration.

5 Conclusions

In this paper, we presented an approach to the use of modularization and atomic decomposition to improve scalability for reasoners, and Chainsaw, a prototype reasoner implementation for this approach. We showed the potential advantages of this approach in the face of increasingly complex and large ontologies, at the expense of pure performance on simple ontologies.

We provided an overview of what trade-offs in terms of memory and speed are involved in our approach, and how our prototype copes with them, presenting a few test results on BioPortal and OBI ontologies, and hinted at future developments and optimizations that have emerged during the prototype creation.

References

1. Cuenca Grau, B., Horrocks, I., Kazakov, Y., Sattler, U.: Modular reuse of ontologies: Theory and practice. JAIR 31, 273–318 (2008)
2. 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)
3. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Extracting modules from ontologies: A logic-based approach. In: Stuckenschmidt, H., Parent, C., Spaccapietra, S. (eds.) Modular Ontologies, Lecture Notes in Computer Science, vol. 5445, pp. 159–186. Springer (2009)
4. Kazakov, Y., Krötzsch, M., Simancik, F.: Concurrent classification of el ontologies. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N.F., Blomqvist, E. (eds.) International Semantic Web Conference (1). Lecture Notes in Computer Science, vol. 7031, pp. 305–320. Springer (2011)
5. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. Automated Reasoning pp. 292–297 (2006)
6. Vescovo, C.D.: The modular structure of an ontology: Atomic decomposition towards applications. In: Rosati, R., Rudolph, S., Zakharyaschev, M. (eds.) Description Logics. CEUR Workshop Proceedings, vol. 745. CEUR-WS.org (2011)
7. Vescovo, C.D., Parsia, B., Sattler, U., Schneider, T.: The modular structure of an ontology: Atomic decomposition and module count. In: Kutz, O., Schneider, T. (eds.) WoMO. Frontiers in Artificial Intelligence and Applications, Frontiers in Artificial Intelligence and Applications, vol. 230, pp. 25–39. IOS Press (2011)