Chainsaw: A Metareasoner for Large Ontologies

Dmitry Tsarkov and Ignazio Palmisano

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

Abstract. In this paper we present the results of running the ORE test suite and its reasoning tasks with three reasoners: Chainsaw, JFact and FaCT++. We describe the newest of these reasoners, Chainsaw, in detail, and we compare and contrast its advantages and disadvantages with the other two reasoners. We show the results obtained and how they compare to each other, and we list the cases in which the reasoners fail to complete a task in the assigned time.

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 modularisation, 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 [10], 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 can use 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 modularisation 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 Modularisation, which are the building blocks of this approach; in Section 3 we describe the implementation details, tradeoffs and strategies adopted, and in Section 5 we present the results on the effectiveness of Chainsaw comparing to JFact and FaCT++ on the ORE test suite.

2 Atomic Decomposition and Modularisation

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 $\widetilde{\alpha}$ the signature of that axiom, i.e. the 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 the set of classes $D \in \widetilde{O}$ such that $O \models C \sqsubseteq D$ (resp. $O \models D \sqsubseteq C, O \models C \equiv D$). 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 [12].

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 [11].

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 $O \models x \equiv y$, 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 $\bot$-locality-based 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 $O \models x \equiv y$, 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 can 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.

This also explains the choice of $\bot$-locality in our approach. It is possible to define $\top$-locality in a similar manner (replace all entities not in signature with $\top$), and use it for the module extraction. Module extraction could be also done in a more complex manner, called $STAR$, interleaving $\top$ and $\bot$ extraction passes until a fixpoint is reached. However, $\top$-local modules are usually larger than $\bot$-local ones, so there is no reason to use them. The $STAR$ modularisation, although provides slightly smaller modules, does not ensure properties from Proposition 1, that are necessary for our approach.

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 optimisations 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 tradeoff 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 least recently used (LRU) caches; this has shown to perform rather well in some of our tests, 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 optimisation 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 optimisation 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.

4 Characteristics of the Reasoners

In this paper we present the comparison between three reasoners that have something in common.

Chainsaw is an OWL 2 DL reasoner that uses modularity to improve query answering for large and complex ontologies. FaCT++ [5] is a tableaux highly optimised OWL 2 DL reasoner implemented in C++. JFact[1] is a Java implementation of FaCT++, that has extended datatypes support. The description of the reasoners' features can be found in Table 1.

Characteristic	Chainsaw	JFact	FaCT++
OWL 2 profile supported	EL, RL, QL, DL		
Interfaces	OWLAPI[2]	OWLAPI	OWLAPI, LISP
Algorithms	AD/sub-reasoner	tableaux	tableaux
Optimisations	meta-reasoning sub-reasoner	same as FaCT++	N/A
Advantages	scalability; good modularity approximation	pure Java; extended DT	good general performance
Disadvantages	AD overhead	work in progress	OWLAPI interface is complicated
Application focus	large ontologies	general purposes	

Table 1. Characteristics of compared reasoners.

Some notes about the table. The algorithm used in Chainsaw for answering hierarchical queries is based on Labelled Atomic Decomposition. For entailment queries an algorithm used in the chosen sub-reasoner is applied. The architecture

[1] http://jfact.sourceforge.net
[2] http://owlapi.sourceforge.net

provides the possibility of using different kinds of reasoners for different queries, so that more efficient reasoners can be used when possible; however, our current implementation does not provide this functionality yet.

We are not going to say much about FaCT++ and JFact here; the optimisations used in FaCT++ are described in a number of publications [6–9], and JFact is a port of FaCT++, so it uses the same techniques and contains the same optimisations. The advantage of Chainsaw is the focus: for every query it uses only the necessary part of the ontology. This allows it to work in situations where the ontology is too large and complex to be dealt with by any other reasoner. So we see the main application area of Chainsaw as large and complex ontologies. Both FaCT++ and JFact are general purpose reasoners, which can be easily integrated in any application that uses `OWLReasoner` interface. In addition to that, FaCT++ could be used directly from C and C++ programs.

Summarising the paper results, the main disadvantage of Chainsaw is the overhead that is brought in by the need to build a LAD and maintain a number of sub-reasoners that answers particular entailment queries. JFact is still a work in progress, so there is a high variance in performance depending on the task. Also, it has a small user base, by which we mean that there are probably bugs which have not been found yet. FaCT++ main problem, when used from Java applications, is its JNI interface, which makes it cumbersome to set up in some environments; for example, in web applications, native libraries are not straightforward to load and work with, and an error might cause issues to the whole application server. The datatype support is not very refined at the time of writing as well.

5 Empirical Evaluation

To check the performance of Chainsaw we ran several tests with it. For the tests we used a MacBook 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++ and JFact v 0.9.

In Table 2 we present the results coming from the ORE test suite, specifically the number of tests that our reasoners failed; these failures are either timeout failures or problems with some unsupported feature of the input ontologies.

As reported in the Total row, Chainsaw has the least amount of failures, while JFact has the highest; this can be explained in terms of time needed for the tasks: we know already that JFact performances are worse than FaCT++, and Chainsaw does not need to perform all tasks on the whole ontology. This gives it an edge over both JFact and FaCT++ in the case of large/complex ontologies; although FaCT++ is much faster on smaller ontologies, larger ontologies push it over our five minutes timeout, thus causing a failure.

We do not report detailed timings for every task in the test suite. Instead, for every task we select the minimal time for successfully completing the task

Task	Chainsaw	JFact	FaCT++
Classification	10	9	21
Class Sat	0	14	3
Ontology Sat	0	11	0
Pos. Entailment	2	3	2
Neg. Entailment	0	0	0
Retrieval	0	1	2
Total	12	38	28

Table 2. Number of failed tests in the ORE test suite.

and use it to calculate a normalised performance index for each reasoner. This provides a relative measure on how good a reasoner is compared to the others.

The index is calculated for the initialisation of a reasoner and for performing the task itself. We then average this index for every reasoner over all the ontologies in every test suite task, and the resulting values are in Table 3.

Task	Init			Main Task		
	Chainsaw	JFact	FaCT++	Chainsaw	JFact	FaCT++
Classification	1.01	3.85	1.75	46.6	43.59	1.32
Class Sat	1.04	2.86	1.74	14.15	126.9	3.9
Ontology Sat	1.05	2.77	1.85	19.17	56.6	1
Pos. Entailment	1	4.18	2.2	51.27	76.2	1
Neg. Entailment	1	4.13	2.06	58.36	119	1
Retrieval	1.04	3.35	1.88	4.39	44.37	25.57

Table 3. Average performance indexes for all reasoners over the ontologies included in each reasoning task.

As can be seen from the data in Table 3, initialisation is a simple process for Chainsaw; in fact, it does very little work at this stage, while both JFact and FaCT++ do a substantial amount of loading and preprocessing; Chainsaw is instead delegating this work until it becomes necessary, i.e., at query answering time.

At query time, FaCT++ turns out to be the fastest reasoner by a large margin; Chainsaw comes second and JFact last. Chainsaw is faster in the retrieval task, but its results were checked manually and they are incorrect; the speed is therefore probably due to a bug in the modularisation code that is used for building the module to be used, and its good performance should not be taken into account.

It is worth mentioning that Chainsaw average includes the ontologies on which the other reasoners failed; therefore, we can conclude that, while slower than FaCT++, it can provide answers in cases in which the other reasoners would not be able to answer in a timely manner at all.

6 Conclusions

In this paper, we presented the relative performances of the three reasoners, and listed their advantages and disadvantages. From these results, we deduce that Chainsaw has the potential of providing acceptable performances where other reasoners fail; it also emerges that its current performances in other cases are not on a par with more mature reasoners such as FaCT++. We consider this an encouraging result, and we presented a few improvements that are already under development.

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. Tsarkov, D., Horrocks, I.: Efficient reasoning with range and domain constraints. In: Haarslev, V., Möller, R. (eds.) Description Logics. CEUR Workshop Proceedings, vol. 104. CEUR-WS.org (2004)
7. Tsarkov, D., Horrocks, I.: Optimised classification for taxonomic knowledge bases. In: Horrocks, I., Sattler, U., Wolter, F. (eds.) Description Logics. CEUR Workshop Proceedings, vol. 147. CEUR-WS.org (2005)
8. Tsarkov, D., Horrocks, I.: Ordering heuristics for description logic reasoning. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI. pp. 609–614. Professional Book Center (2005)
9. Tsarkov, D., Horrocks, I., Patel-Schneider, P.F.: Optimizing terminological reasoning for expressive description logics. J. Autom. Reasoning 39(3), 277–316 (2007)
10. Tsarkov, D., Palmisano, I.: Divide et impera: Metareasoning for large ontologies. In: Proc. of 9th International Workshop OWL: Experiences and Directions (OWLED 2012). To Appear (2012)
11. 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)
12. 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)