Ontology Performance Profiling and Model Examination: First Steps

Taowei David Wang[1], Bijan Parsia[2]

[1] Department of Computer Science, University of Maryland,
College Park, MD 20742, USA,
tw7@cs.umd.edu
[2] School of Computer Science, University of Manchester, UK
bparsia@cs.man.ac.uk

Abstract. "[Reasoner] performance can be scary, so much so, that we cannot deploy the technology in our products." – Michael Shepard[3]. What are typical OWL users to do when their favorite reasoner never seems to return? In this paper, we present our first steps considering this problem. We describe the challenges and our approach, and present a prototype tool to help users identify reasoner performance bottlenecks with respect to their ontologies. We then describe 4 case studies on synthetic and real-world ontologies. While the anecdotal evidence suggests that the service can be useful for both ontology developers and reasoner implementors, much more is desired.

1 Introduction

Ontology engineering is the discipline of building a certain class of computational artifacts — ontologies — which typically are a component of a larger software system. Ontologies are used as conceptual models, for data integration, or to directly represent information in a variety of domain areas. Today, most self-described ontology development has used logic-based representation languages to express ontologies, that is, ontologies are (in part) *theories in some logic*. Indeed, the Web Ontology Language (OWL) is based on a description logic and has spurred a large rise in the number of publicly available logic based ontologies[15].

However, as with most interesting logics, the reasoning services which are critical for the development and sometimes the deployment of ontologies have very bad worst case complexity. For example, consistency checking for $\mathcal{SHOIN}$, the DL underlying OWL is NEXPTIME complete. While modern reasoners (such as FaCT++, KAON2, and Pellet) employ an increasingly sophisticated array of optimizations, it is still not particularly difficult to stymie them. When the reasoners seem, somewhat randomly, to never halt while performing reasoning services, users often feel lost, frustrated, and helpless. Most do not have the training or the expertise to delve into their reasoner to figure out what is going

[3] `http://lists.w3.org/Archives/Public/public-owl-dev/2007JanMar/0047.html`

on, even if the source code is available. Consequently, users try to (1) Remove portions of the ontology that are thought to be the causes of reasoning performance problems, or (2) contact reasoner implementors for a detailed explanation. However, precisely because of the lack of expertise, attempts to remove axioms in order to reduce the computational load often results in efforts a little better than blind guesses. Though experts such as reasoner implementors have general ideas of the type of axioms that can create performance problems, general guidelines may not help solving specific problems. In addition, it can be difficult to identify performance problems due to hard-to-handle logical axioms when there are many easy-to-handle axioms. The lack of tool support and theory in this area is the primary reason that even experts find it a significant challenge to explain the performance of their reasoner against certain ontologies.

The situation in ontology engineering with regard to performance tuning is similar to that with regard to debugging a few years ago. In the past two years or so the state of the tooling for finding and explaining semantic errors in ontology has gone from nothing to respectable [12][11]. The availability of robust services and tools for debugging ontologies allows ontology engineers to build larger, more correct, more interesting ontologies in less time and with less expertise and tedium. Today, performance analysis of ontologies against reasoners is a painful, tedious, manual affair even to identify a bottleneck as a test case for reasoner implementors.

In this paper we take some first steps toward supporting the performance analysis of description logic knowledge bases. In particular, we focus on understanding the effort in testing the *satisfiabilty* of a class using a tableau reasoner. In the following sections, we relate our research to similarities in other disciplines, describe our prototype tool and case studies on how the tool relays the internal states of a reasoner to help a user understand the performance bottleneck.

2 Background and Challenges

We draw the analogy between software engineering and ontology engineering. Both ontologies and software source code are human-written computational artifacts meant to be processed by other programs for a purpose. Source code is to be compiled and run in an environment, and ontologies are to be processed by reasoners for entailments. Software profiling involves collecting various performance statistics during program execution which are correlated with statements in the source code. Similarly, we envision an ontology profiler gathering performance related statistics during reasoning and correlating them with axioms (or terms) in the ontology. We emphasize that we are not interested in profiling the reasoners as programs themselves. We are treating the reasoners as fixed entities, and instrument their behaviors with regards to different parts of an ontology to identify sources of performance bottlenecks. Of course, just as sometimes one has to look to the behavior of the compiler, interpreter, query engine, or runtime libraries to actually solve a performance problem, sometimes the problem can only reasonably be solved by investigating and modifying the reasoner. However, even in these cases, it can be very helpful if the users can effectively isolate the aspects of their ontology that is causing problems.

In [10], Jeffery identifies four challenges to building successful program monitors (e.g., for debugging or performance analysis), all of which apply to the ontology engineering case:

1. **Volume of Data**: The ontologies themselves can be large and complex enough to require significant tool support even in their asserted form. The search space for reasoning services for expressive description logics is very large (as indicated by their EXPTIME to NEXPTIME worst case complexity). Tableau reasoners build finite graph representations (completion graphs) of models of the given ontologies, and this graph and its construction trace can be large and unwieldy. The challenge here is first deciding what are the most useful data, and subsequently, how to allow users explore and correlated the two data sources to gain insight.
2. **Dimensionality of Data**: As the inference services proceed, data from tracing the reasoners' high-level behavior and the resultant completion graphs are generated. At each time point during the execution, the state of the reasoner can be described by the (incompleted) completion graph. The completion graph does not only contain structural information, but also sets of categorical information as labels. Additional flags (e.g. blocked, cached) designating techniques used by the reasoner also add dimensions to the data. Finally, statistics for raw performance measures should also be collected as performance overviews.
3. **Intrusion**: Software monitors typically must alter the program and the environment in order to gather useful data. Instead of providing execution-time monitoring, which may impose unrealistic slowdown, we employ less invasive "postmortem" methods to allow users to examine the performance data after the execution.
4. **Access**: Many aspects of the execution of inference services are not usually accessible. Users typically use reasoners as an oracle. They ask whether a certain concept is satisfiable, and the reasoner returns "yes" or "no". There is little information available for the user to review. Exposing the graphs and the internals of the reasoners' operations is not a straight-forward task, however. The non-deterministic nature of the tableau algorithms and its optimizations may cause the inference services to behave differently when given the same ontology. How to present these differences and still keep a coherent picture is a challenge. While reasoners can handle many ontologies within a reasonable time frame, sometimes the inference services never seem to end, or all available memory is consumed. When this happens, often no information can be given back to the user. To be able to handle such cases, even at a preliminary level, is key.

In program optimization, we can distinguish between two basic sorts: macro or "algorithmic" or "design" improvements, and micro or "code level" improvements. Changing data structures for one better suited to the problem is an example of the former. Loop unrolling is an example of the latter. Obviously, there is a continuum between these poles, and some activities — such as tuning the garbage collector — do not obviously fit. It's not clear, exactly, whether these categories provide a useful framework for thinking about ontologies. There are certainly analogous practices to micro-optimization: sometimes twiddling the axioms can have a large beneficial effect on performance. Many in-reasoner optimization involve transforming axioms into a better form for the reasoner. Similarly, the practice of approximating an ontology originally developed in a more expressive logic in a less expressive logic (see Dolce Lite[4] which is an OWL DL version of a full first order logic based ontology) or approximating difficult constructs (such as approximating nominals as new disjoint atomic classes) can be seen as something of a macro-optimization, at least, as an attempt to macro-optimize. Obviously, as with software engineering, it is important to understand what is lost with such changes. Not all optimizations preserve the exact behavior of the original program. Similarly, some changes to ontologies will not respect the intended representation of the domain and even lose significant entailments. Such are the compromises users face.

One significant point of potential disanalogy is that reasoning techniques vary widely. While tableau algorithms are still the dominant form of reasoning with OWL DL, other techniques such as reduction to disjunctive datalog[8] and reduction to first order logic, such as Hootlet[5]and MSPASS[9], are significantly different in most details. Not only is there different information to extract, but the behavior model, thus how to fruitfully interpret the extracted data, is very different. The techniques we explore in this paper were developed for a specific reasoner, Pellet, and thus for a fairly bog-standard tableau reasoner. This obviously is limited, but is not unreasonably specific. Even if the techniques would not translate directly to other tableau reasoners, it is important to determine *whether such services* are useful to ontologists, and how.

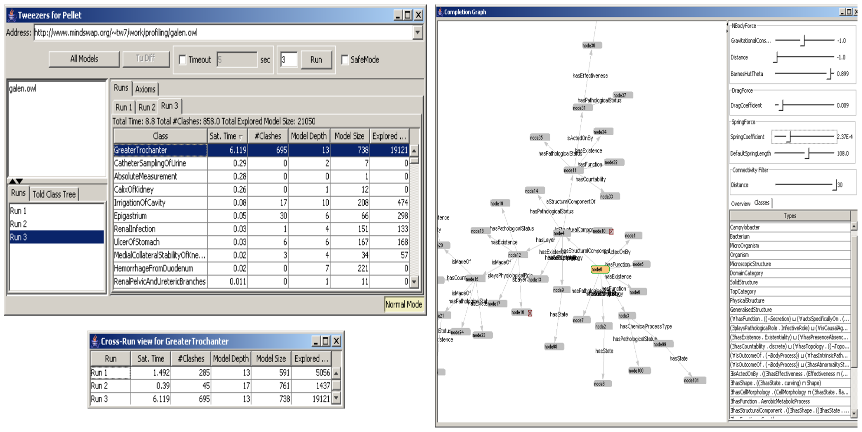
3 Tool Design and Implementation

Tweezers is a prototype utility that instruments Pellet and allows users to gain access to the inference results and performance statistics. The main interface (See Figure 1) enables users to view and sort a set of performance statistics. We collect the following statistics: *Sat. Time*: The CPU time it takes to perform satisfiability check for a particular class. *#Clashes*: The number of clashes encountered when performing the satisfiability check. This measures how many dead ends reasoner run into before finding a completion. *Model Depth*: The depth of the completion graph. *Model Size*: The size of the completion graph. *Explored Size*: Number of nodes generated but were not in the final completion graph (due to clashes and backjumping). This is a rough measure of "wasted effort". Upon loading an ontology from the Web, Tweezers automatically performs sat-

[4] http://www.loa-cnr.it/DOLCE.html

[5] Hoolet: `http://owl.man.ac.uk/hoolet/`

isfiability checks on all classes. The default sat check behavior can be modified via several controls, described below.



Address: http://www.mindswap.org/~tw7/work/profiling/galen.owl

Timeout 5 sec 3 Run SafeMode

galen.owl

Run 1 Run 2 Run 3

Total Time: 8.8 Total #Clashes: 858.0 Total Explored Model Size: 21050

Class	Sat. Time	#Clashes	Model Depth	Model Size	Explored ...
GreaterTrochanter	6.119	695	13	738	19121
CatheterSamplingOfUrine	0.29	0	2	7	0
AbsoluteMeasurement	0.28	0	0	1	0
CalixOfKidney	0.26	0	1	12	0
IrrigationOfCavity	0.08	17	10	208	474
Epigastrium	0.05	30	6	66	298
RenalInfection	0.03	1	4	151	133
UlcerOfStomach	0.03	6	6	167	168
MedialCollateralStabilityOfKne...	0.02	3	4	34	57
HemorrhageFromDuodenum	0.02	0	7	221	0
RenalPelvicAndUretericBranches	0.011	0	1	11	0

Cross-Run view for GreaterTrochanter

Run	Sat. Time	#Clashes	Model Depth	Model Size	Explored ...
Run 1	1.492	285	13	591	5056
Run 2	0.39	45	17	761	1437
Run 3	6.119	695	13	738	19121

NBodyForce
GravitationalCons... -1.0
Distance -1.0
BarnesHutTheta 0.899
DragForce
DragCoefficient 0.009
SpringForce
SpringCoefficient 2.37E-4
DefaultSpringLength 108.0
Connectivity Filter
Distance 30

Types
Campylobacter
Bacterium
MicroOrganism
Organism
MicroscopicStructure
DomainCategory
SolidStructure
TopCategory
PhysicalStructure
GeneralisedStructure
(∀hasFunction . ((¬Secretion) ⊔ (∀actsSpecificallyOn . (...
(∃playsPathologicalRole . InfectiveRole) ⊔ (∀isCausalAg...
(∃hasExistence . Existentiality) ⊔ (∀hasPresenceAbsenc...
(∃hasCountability . discrete) ⊔ (∀hasTopology . ((¬Topo...
(∀isOutcomeOf . (¬BodyProcess)) ⊔ (∀hasIntrinsicPath...
(∀isOutcomeOf . (¬BodyProcess)) ⊔ (∃hasAbnormalitySt...
∃isActedOnBy . ((∃hasEffectiveness . (Effectiveness ⊓ (...
∃hasShape . ((∃hasState . curving) ⊓ Shape)
∃hasCellMorphology . (CellMorphology ⊓ (∃hasState . fla...
∃hasFunction . AerobicMetabolicProcess
∃hasStructuralComponent . ((∃hasShape . ((∃hasState . ...

Fig. 1. On the left side is the Tweezers main interface. The statistics are sorted by Sat. Time. A cross-view of the statistics for a class is shown at the bottom. The right side shows the interface for completion graph inspection. The graph is visualized on the left. The right side contains the visualization control, and labels for the currently selected node (highlighted in orange)

For convenience, users can execute several runs in batch fashion with one click. Statistics from multiple runs let users better deal with nondeterminism via averages. Viewing of statistics from different runs is managed by tabs. Double-clicking a row gives users a view of the statistics of that class across all the runs. Double-clicking again here launches the completion graph viewer (See Figure 1). The completion graph is shown as a network in force-directed layout, using the open-source Prefuse library [4]. Each node represents an individual, and each edge represents a set of roles between the two individuals. The nodes are numbered in the order that they are generated. In a panel at the bottom right part of the interface, the labels of the currently selected individual are displayed in the order that they are added. Nodes with special attributes are additionally decorated: the first node is outlined in green, and blocked nodes are shown with a blocked symbol in red.

By clicking on "All Models", users can perform a run where Pellet finds all possible representations of models for each class. Satisfiability checks typically stop when a first model is found. Here we force Pellet to find all models. These models are represented by a set of completion graphs, and these graphs are available for inspection using the same interactive visualization interface. Interested users can explore these completion graphs to look for qualitative differences.

With each run, users can optionally set a timeout limit (default is 5 seconds) for how much time Pellet is allowed to spend performing satisfiability check for each class. When the timeout is reached, Pellet stops processing that class, and moves onto the next, no statistics are kept for the aborted class.

The statistics and completion graphs are normally kept in-memory for faster access. The size of completion graphs for classes in some ontologies can be in the order of thousands, consuming all memory so that Tweezers fails. To better make use of main memory, Tweezers has a “Safe Mode” that allows the storage of collected data to be made on hard drives. In this mode, maximum amount of memory is ensured for the satisfiability check for every single class. When encountering an out-of-memory error in normal mode, Tweezer will attempt to recover and restart the process in Safe Mode.

Tweezers is built as part of pre-released version of Pellet 1.4 (no OWL 1.1 support). Currently it provides instrumentation for ontologies of expressivity $\mathcal{SHIN}(\mathcal{D})$ (OWL-DL without nominals). However, this is mostly a problem simplification decision, not a technical barrier. This Pellet is bundled with a version of Swoop, which contains useful utilities such as ontology modularization that can additionally help isolate performance problems. These tools can be found here [6]

4 Case Studies

We describe a series of case studies on how our prototype tool help identifying the causes of performance bottleneck. There are four case studies, presented here roughly in the chronological order that they were studied.

4.1 Galen, Round 1

One common “difficult” OWL ontology is the version of the Galen[7] medical ontology translated from the original GRAIL formalism by Ian Horrocks, and has over 2700 classes, 400 ObjectProperties, and 300 GCIs. It is in the DL expressivity $\mathcal{SHF}$. This is an interesting initial case because of the large number of GCIs. We expected to see a lot of variation in the performance statistics when we run satisfiability checks in Pellet. We were interested in finding out if there are consistently “difficult” classes, and if we could fine-tune it.

We initially ran Tweezers on Galen a few dozen times, and ascertained that the class *GreaterTrochanter* consistently has the worst statistics of all classes. The average completion graph size for *GreaterTrochanter* was well over 600, by far the largest in this ontology. We traced a few of the graphs manually, attempting to see if there are axioms that can be added or removed to reduce the complexity of the completion graph. In the graphs, we found that more than

[6] `http://www.mindswap.org/~tw7/work/profiling/code/index.html`

[7] Galen: `http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/galen.owl`. It is important to note that this is a translation of a very old version of Galen (sometimes called “not-Galen”). The current production version of Galen is orders of magnitude larger and is not classifiable by any existing reasoner, see: `http://www.co-ode.org/galen/index.php`

one individual is both a *Femur* and a *Tibia*, which seemed counterintuitive. We thought that by making *Femur* and *Tibia* disjoint, we may be able to reduce the completion graph size and make Pellet work less hard. After consulting the on-line version of Gray's Anatomy [8] to make sure our intuitions is not wrong (after all, we are not the domain experts!). The classes *Femur*, *Tibia*, *Fibula*, *Humerus*, *Ulna*, and *Radius* are direct subclasses of *LongBone*. We made an additional version of Galen, where these *LongBone*s are all pairwise disjoint so we can compare the effects of disjoint axioms applied in a surgical setting as opposed to a more general setting.

We then ran Tweezers 5 times on each of the 3 versions of Galen: original, Femur-Tibia-Disjoint, and LongBones-Disjoint. The averages of the statistics are summarized in Table 1.

Table 1. Averages of performance statistics over 5 runs each for the 3 versions of Galen. The left columns shows the statistics for performing satisfiability check for the class *GreaterTrochanter*, while the right columns show the same statistics for performing satisfiability for all classes in the ontology.

Scope	GreaterTrochanter					Ontology		
Statistics	Time	Clashes	Depth	Size	Explored	Time	Clashes	Explored
Original	1.3	118.6	12.8	689.2	5814.4	5.94	322.2	7892.2
F+T Disjoint	0.66	95	9.8	396.6	3184	3	297.4	5528.8
LB Disjoint	0.15	16	8	373.4	601.8	2.78	224.8	2756.8

Satisfiability check time for the *GreaterTrochanter* improved roughly 50%. The same statistic also improved roughly 50% for the all-class case. The number of clashes for GreaterTrochanter is reduced by 45%. The completion graph size is also reduced from routhly 700 to about 400. In LongBones-Disjoint, the satisfiability check time for *GreaterTrochanter* further improved so that it is an oder of magnitude faster than the original. The number of explored nodes for GreaterTrochanter is greatly reduced (from 3184 in the Femur-Tibia-Disjoint to 601). However, the other improvements are too minor to be considered of any importance.

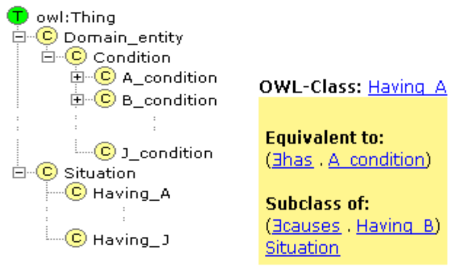
4.2 Causal Chains

Modeling causal relationships for diseases and diagnosis is common in biomedical ontologies. The Galen team in Manchester has devised a small test ontology to see if a particular way of modeling can capture the intended knowledge and allow reasonable performance in reasoners for real applications. This Causal-Chain ontology contains 43 classes, 4 object properties (2 pairs of inverses: *has*, *is_had_by*, *causes*, *is_caused_by*), and has the DL expressivity $\mathcal{ALCI}$. The told class structure has two main branches. The first branch is a simple hierarchy

[8] Gray's Anatomy: `http://education.yahoo.com/reference/gray/`

of *Conditions*. The second branch contains the class *Situation* and its list of children (See Figure 2). Each child of *Situation* is named *Having_X*, where *X* corresponds to a mirror class *X_Condition* in the first branch. Generally, every *Having_X* have the following axioms:

1. $Having\_X \equiv \exists has.X\_Condition$
2. $Having\_X \sqsubseteq \exists causes.Having\_Y$
3. $Having\_X \sqsubseteq \exists Situation$



OWL-Class: Having_A

Equivalent to:
(∃has . A_condition)

Subclass of:
(∃causes . Having_B)
Situation

Fig. 2. On the left side, an abbreviated version of the class tree is shown. On the right hand side, detailed class definition for *Having_A* as is shown in Swoop.

where *Y* stands for the letter that occurs one after *X* (See Figure 2). The implication is that every *Having_X causes* a *Having_Y*, and this causal chain goes from *Having_A* to *Having_J*. We were told that FaCT++ was able to classify this ontology in seconds, but Pellet could not classify this ontology. We theorized that the causal chain was the problem – since the ontology is very small, and the structure is regular and simple. We would like to make causal chains of different length, and see which ones Pellet can process. Using Swoop, we modularized the ontology (as described in [3]) along the causal chain, once for every *Having_X* class. We end up with 10 modules of increasing complexity, from *Having_J* to *Having_A*. Pellet can classify all module except the ones for *Having_B* and *Having_A*. Completion graphs revealed that there were many unexpected disjunctions in the nodes' labels. Moreover, Pellet debugging message showed that the completion rules for domain axioms are being fired. The only problem was that there were no domain axioms in the ontology.

We later determined that the GCIs in this ontology are handled by a reasoner optimization called "role absorption". The GCIs 4. and 5. are transformed into the domain axioms 6. and 7. by role absorption [14]:

4. $\exists has.Condition\_X \sqsubseteq Situation$ (from 1. and 3.)
5. $\exists has.Condition\_X \sqsubseteq \exists causes.Having\_Y$ (from 1. and 2.)
6. domain(has, $Situation \sqcup \neg\exists has.Condition\_X$ (from 4.)
7. domain(has, $\exists causes.Having\_Y \sqcup \neg\exists has.Condition\_X$ (from 5.)

If there are n such domain restrictions, every node that had a *has* successor would be labeled with at most n complex domain types, where each type contained a disjunct of 2 class expressions, for a total of 2^n possible choices. From *Having_A* to *Having_J*, there were 19 such GCIs (*Having_J causes* nothing), thus we had a search space of cardinality 2^{19} for *every* node that generated a *has* relation. To make matters worse, the presence of inverse properties means that more sophisticated blocking techniques needed to be used in the tableau to ensure the correctness of the inferences [7]. As a result, the size of completion graphs are usually much larger, and inference takes longer.

We found two possible ways to tune it for better performance. The first one is to remove the inverse assertions. The second one is to change the the $\equiv$ assertions from the *Having_X* classes to $\sqsubseteq$. Both of these techniques reduced the classification time for Pellet from impossible to within seconds. The first technique lowers the expressivity of the language used so that the reasoner could block more easily, reducing the amount of work. In the second case, removal of the equivalence axioms means we get rid of the GCIs, and no role absorption would occur. Though both solutions makes the ontology manageable, they also come with a price. In the first case, if inverses were the central focus of modeling, then it would be unwise to remove them. On the other hand, if the intended use of the ontology is for instantiation (determining what classes individuals are members of), then the second method would not be appropriate. This decision should be made by the ontology creators and application developers.

4.3 LKIF-Core Ontology

The LKIF is a suite of OWL ontologies that describe the legal domain.[9] A merged version of a snapshot of the ontology from late Feburary 2007 is used in this discussion, and can be found here [10]. The ontology is not large, containing 206 classes, 1 data type property, 106 object properties, and no individuals. It has the DL expressivity $\mathcal{SHIN}(\mathcal{D})$. There are 75 inverse properties. Neither FaCT++ (called from Protégé 4.27) nor Pellet was able to classify this ontology in reasonable amount of time. Desires to find out more about the behaviors of the reasoner with respect to the ontology was expressed in the Pellet user mailing list. In particular, the user would like to know whether there is a specific construct or pattern that has made the ontology unprocessable[11].

In the Causal-Chain case study, it was easy to isolate the potential problems by manually identifying the difficult part of the ontology because of its size and regularity in structure. We then modularized the ontology to examine the effects of the length of causal chain on performance. However, it is not so obvious here. Manually inspecting the asserted axioms showed that GCIs formed through equivalence axioms and subclass axioms in a hierarchy (as in the Causal-Chain case) were abundant. Class definitions also use the inverse properties frequently.

[9] LKIF-Core ontology web page: `http://www.estrellaproject.org/lkif-core`
[10] `http://www.mindswap.org/~tw7/work/profiling/others/lkif-all-correct.owl`
[11] `http://lists.owldl.com/pipermail/pellet-users/2007-February/001257.html`

We expected that much of the ontology will be problematic for the reasoner. However, blindly modularize the ontology did not seem a fruitful method. We used the timeout feature of Tweezers to restrict the time Pellet can use to perform satisfiability check for a particular class. The timeout was originally set to be 5 seconds. As expected, majority of the classes could not be checked for satisfiability within the timeout(151 out of 206). Of the ones that could be, 17 were unsatisfiable. The satisfiable classes all had fairly small models, and took no more than 0.1 second to perform satisfiability check, and were without clashes. In this case, very little useful information was available to guide us to further isolate the problems. By extending the timeout limit to 10 seconds, we were able to see Tweezers process 90 classes (including the 17 unsatisfiabile ones). Of these classes, many have completion graphs of the same size and depth, indicating that they may be mostly the same, sharing a common, large structure. However, the size of such graphs (over 3000+) made detailed manual inspections prohibitive. Instead, we suggested to the users to remove non-critical equivalence axioms and make less important properties not inverse, symmetric, or inverse functional. Pellet could process the ontology within seconds once these property attributes were removed.

There are a few reasons why this was an interesting case study. One, the situation that users get frustrated with reasoner's performance with respect to their own ontologies was consistent with our expectation, especially when the use of reasoners is not part of the ontology development cycle (as suggested by the existence of many unsatisfiable classes). Secondly, more sophisticated users of the OWL language desire to gain more understanding of their own ontology when this situation arises, and only finds (1) the current tools are not easily amendable to exposing the internal states and histories, and (2) when an ontology is unprocessable as a whole, the users get nearly no information about their ontology (even though part of it is processable). Our prototype tool attempted to give users more feedback, though in this particular case, only generic advice could be rendered without systematic examination and experimentation of large number of "difficult" classes.

4.4 Galen, Round 2

The above case studies showed that having inverse properties in even small, but expressive ontologies can have far-reaching performance consequences. We returned to the ontology Galen and performed an experiment by adding an inverse property. There are two main object property branches in Galen. One is rooted at *DomainAttribute*, while the other is rooted at *InverseDomainAttribute*. These two branches mirror each other, and each branch has about 200 properties. Though the name suggests that the properties in one branch would be inverses of the corresponding ones in the other, no actual inverse assertions exist. For our experiment, we selected one pair of corresponding object properties to be inverses of each other: *ActsOn*, *isActedOnBy* . These two properties were used in definitions of 61 classes. When looking at the inferred hierarchy of Galen, there were a total of 385 classes effected by these definitions because they were descendants of the 61 classes. Pellet was not able to classify this ontology. Running Tweezers with a 5 second timeout showed that satisfiability of 900+ of the 2700+ concepts could not be checked. Our friend GreaterTrochanter belonged to that difficult group. In the original Galen, only GreaterTrochanter had any notable size in its completion graph. In this single-inverse version, many classes have completion graphs of sizes over 10k. Even the simple classes that did not seem to have any connection to the inverse properties would have sizes in the order of thousands!

5 Discussion

These case studies exhibited all the facets of the research problem. First, there is a real need from the user community. Instrumenting the reasoner and allow users to view the performance statistics is one way to help them identify the problem which, aside from helping them cope with the particular problem, can increase their overall satisfaction with the process. We received very positive feedback from users about our analyses and they were very vocal about obtaining access to Tweezers. It seems that even if modifications to their ontologies would not be acceptable, simply *knowing* what the problem qualitatively improves their experience. It is shown in the first case study that when the ontology can be classified by a reasoner, it is possible to fine-tune the ontology via adding or removal of axioms to improve the reasoning performance. We showed that a strategically placed disjoint axiom can greatly reduce the performance time for both a specific concept and the entire ontology. The intuition is that the disjoint axioms restricts the reasoner from adding labels, thus limiting the size of the completion graph, and possibly pruning search space. However, it is possible that too many disjoint statements may cause performance problems, as this may restrict the reasoner too much, and the overhead for many backtracks catches up . We conjecture that there is a “right amount” of disjoints that can optimize the performance for many ontologies. We are currently investigating methods of determining such “right amount”.

In small ontologies with very regular structures such as the Causal-Chain ontology, manual inspection of the axioms was often enough to have an inkling of what might be causing the problem. Modularizing the ontology proved to be an effective method to simplify the problem and identify the bottleneck. In cases like LKIF with 5-second timeouts, the classes were either so easy that their performance statistics are of no insight, or that the classes were so difficult that no useful information can be collected, finding what classes with respect to which to modularize is a challenge. The LKIF case study also revealed that large completion graphs are prohibitive for both the tools to display and humans to digest. More automated methods should be used to look for points of interest in the completion graph.

The final 3 case studies showed that the presence of inverse properties can drastically change reasoner performance. They revealed how inefficiently inverses are handled. Currently, Pellet chooses a strategy for performing inference services by the expressivity of the ontology. For example, when an ontology contains inverse properties, a more sophisticated blocking technique is employed, and this technique is used for all inference tasks for the entire ontology. However, it may be the case that there is a large part of the ontology that does not use inverses (such as in our last case study), and can use a simpler blocking technique without compromising the correctness of the inference. This may be achieved by investigating the possibility of performing modularization for each class prior to satisfiability check and use the expressivity of such module instead of the ontology, or having the reasoner dynamically change the strategy while reasoning.

6 Related Work

To our knowledge there is has been no previous work on providing *ontologists* with tools which attempt to explain the reasoning time characteristics of description logic reasoners in any detail. Reasoners like FaCT++ and Pellet can be configured to provide some feedback during the reasoning process, but this is generally limited to fairly coarse grain timing information (for example, such as would be presented in a progress bar). Sometimes, there is some correlation presented between time and certainly terms (e.g., a reasoner may print start and end points for testing the satisfiability of a particular class), but never to particular axioms or to particular internal behaviors (e.g., backtracking). Also, reasoners tend to be rather unforgiving if there is a memory or timeout problem with an ontology. For example, if a single subsumption test runs out of memory, FaCT++, Pellet, and Racer simply abort the entire classification process (sometimes aborting the the entire session and requiring a restart of the reasoner). As users and developers of reasoners and ontology development environments for several years, we were surprised that it had not occurred to us that this extremely user-hostile behavior was not a good idea. This all-or-nothing approach is deeply embedded in the culture even though, in principle, many of the subsumptions are easy to compute and are accessible, albeit in a clumsy way, from the various APIs. Just lifting this draconian behavior is useful, although raises interface challenges on how to present the resource failures.

Ontology development environments such as Swoop and Protégé provide a syntactic analysis of ontologies designed to help users gain some sense of the overall complexity of an ontology. These analysis include statistics on the number of classes, properties, and individuals and the number of GCIs, inverse properties and the like. They also attempt to classify more precisely the description logic the ontology falls into. This feature first appeared in Swoop and was inspired by the "species validation" service defined by the OWL specifications. The species of OWL — Lite, DL, Full — were intended to give a very course grained idea of the "difficulty" of reasoning with an ontology, all other things being equal. this difficulty is based on the worst case complexity and general experience with reasoning with the corresponding logics. Given that the species' worst case complexity range from EXPTIME to undecidable, and that there is considerable expressive overlap between the species, it is clear that they are not a particularly helpful guide to expected performance, though they have been used as such. Recently, there has been a renaissance in the area of description logics with tractable (by some measuers) worst case complexity. These logics in some sense promise better scalability (and their novel inference techniques have cracked ontologies that current tableau reasoners fail to handle), although this can vary significantly in particular cases. They also involve significant expressivity compromises. The idea of "light weight species" has considerable appeal to users as shown by the enthusiasm generated by the OWL 1.1 "tractable fragments" document.[12]

There is a growing literature on description logic reasoner benchmarking, though the majority is embedded in discussions of optimizations or new reasoning techniques, e.g.[6]. There is a line which extends from the modal logic community's attempt to generate "hard modal formulae"[5]. Unfortunately, relating these benchmarks to analysis of difficult ontologies (for certain reasoners) has not been systematically attempted.

Model and proof extraction and presentation has a number of uses from education[1] to debugging missing entailments[13]. One of our long term goals is to explore support for *model oriented* ontology development [2]. One reason to focus on the use of model/tableau visualization and exploration in a profiling context is that we do not need domain experts in order to perform useful experiments. Subjects can be set tasks such as "improve the performance of satisfiability testing while minimizing the loss of subumptions" and model generation and exploration tools without them needing to understand the subject matter of the ontology. In this way, we believe that performance tuning has methodological value independently of its substantive value to users.

7 Future Work and Conclusions

Through our case studies, we have demonstrated the difficulty of ontology performance profiling, and how our prototype implementation can help. However, these case studies also revealed the limits of our current tool. Most glaringly, tools and users get easily overwhelmed by multiple, very large completion graphs. The development of more sensible user interfaces, coupled with a more complete coverage of reasoner behaviors (such as axiom-level profiling and completion graph construction history) for exploration and analysis of completion graphs is critical. We also need to study how experts use these tools so the process can be more automated, and more explanations can be given to the average OWL users. Indeed, if the performance is to be *scary*, it should not be scary because of the *unknown.*

[12] `http://www.w3.org/Submission/2006/SUBM-owl11-tractable-20061219/`

8 Acknowledgments

This work was supported in part by grants from Fujitsu, Lockheed Martin, NTT Corp., Kevric Corp., SAIC, the National Science Foundation, the National Geospatial Intelligence Agency, DARPA, US Army Research Laboratory, and NIST. Special thanks to Evren Sirin for his invaluable discussions and help.

References

1. M. D'Agostino and U. Endriss. Winke: A proof assistant for teaching logic. *Proceedings of the First International Workshop on Labelled Deduction, 1998*, 1998.
2. C. Fermller, A. Leitsch, and G. Salzer. Automated model building as future research topic.
3. B. C. Grau, I. Horrocks, Y. Kazakov, and U. Sattler. A logical framework for modularity of ontologies. *Proceedings of the Twentieth International Joint Conference on Artificail Intelligence (IJCAI-2007)*, 2007.
4. J. Heer, S. K. Card, and J. A. Landay. Prefuse: a toolkit for interactive information visualization. *Proceedings of the Conference On Human Factors In Computing Systems (CHI05)*, 2005.
5. J. Hladik. A generator for description logic formulas. *Proceedings of the International Workshop on Description Logics, 2005*, 2005.
6. I. Horrocks and P. F. Patel-Schneider. Comparing subsumption optimizations. In *Proc. of the 1998 Description Logic Workshop (DL'98)*, volume 11 of *CEUR (http://ceur-ws.org/)*, pages 90–94, 1998.
7. I. Horrocks and U. Sattler. Optimised reasoning for shiq. *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI 2002)*, 2002.
8. U. Hustadt, B. Motik, and U. Sattler. Reducing shiq description logic to disjunctive datalog programs. *Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (KR2004)*, 2004.
9. U. Hustadt and R. Schmidt. Mspass: Modal reasoning by translation and first-order resolution. In *Automated Reasoning with analytic Tableaux and Related Methods.* 2000.
10. C. L. Jeffery. *Program Monitoring and Visualization.* Springer, 1999.
11. A. Kalyanpur, B. Parsia, E. Sirin, and B. Cuenca-Grau. Repairing unsatisfiable concepts in owl ontologies. *Proceedings of the 3rd European Semantic Wen Conference (ESWC) 2006*, 2006.
12. A. Kalyanpur, B. Parsia, E. Sirin, and J. Hendler. Debugging unsatisfiable classes in owl ontologies. *Journal of Web Semantics*, 2005.
13. D. McGuinness. *Explaining Reasoning in Description Logics.* PhD thesis, New Brunswick, New Jersey, 1996.
14. D. Tsarkov and I. Horrocks. Efficient reasoning with range and domain constraints. *Proceedings of the 2004 Description Logics Workshop*, 2004.
15. T. D. Wang, B. Parsia, and J. Hendler. A survey of the web ontology landscape. *Proceedings of the 5rd International Semantic Web Conference (ISWC) 2006*, 2006.