Reintroducing CEL as an OWL 2 EL Reasoner*

Julian Mendez and Boontawee Suntisrivaraporn

Theoretical Computer Science, TU Dresden, Germany
{mendez,meng}@tcs.inf.tu-dresden.de

Abstract. The CEL system is known for its scalability of reasoning in the lightweight DL $\mathcal{EL}^{++}$ which has been proved suitable for several ontology applications, most notably from the life science domain. Recently, the DL $\mathcal{EL}^{++}$ has been adopted as the logical underpinning of the OWL 2 EL profile of the new Web Ontology Language which potentially attracts new folks of CEL's users. To seamlessly integrate the reasoner to the OWL user community, we have implemented the OWL API for CEL. This paper describes the challenges, design decision and architecture of this implementation. Additionally, we present experimental results which highlight the scalability of the reasoner, as well as demonstrate a low overhead of our OWL API implementation.

1 Introduction

The system CEL[1] has been a first step toward realizing the dream of a DL system that offers both sound and complete polynomial-time reasoning and expressive means that allow its use in real-world applications. Since it first came into existence in 2005 [BLS05], CEL was the only academic DL system that was capable of classifying entire SNOMED CT. This has subsequently sparked interest in the DL community to research on optimization techniques specific to biomedical ontologies (in particular, to SNOMED CT), and later enabled tableau-based reasoners like FaCT^{++} and RacerPro to take advantage of simple structures of ontologies of this kind. Some of the most effective optimizations employed in these systems are described in [HT05, HMW08]. Besides, the OWL reasoner Pellet, together with the ontology-specific reasoning system Snorocket [Law08], has also implemented the polynomial-time reasoning algorithm which helps boost the performance of reasoning whenever the ontology under consideration is described by a tractable fragment of OWL.

Recently, the OWL working group (WG) has identified three profiles (i.e. logical fragments of OWL 2), which are OWL 2 EL, OWL 2 QL, and OWL 2 RL.[2] The OWL 2 QL profile includes many of the main features of conceptual models such as UML class diagrams and ER diagrams. The OWL 2 RL profile is aimed at applications that require scalable reasoning without sacrificing too

* Funded by the German Research Foundation (DFG) under grant BA 1122/11-1.

[1] CEL's sources are now open and obtainable at `http://cel.googlecode.com`.

[2] `http://www.w3.org/TR/owl2-profiles`

much expressive power. The OWL 2 EL profile is heavily based on the tractable Description Logic $\mathcal{EL}^{++}$, as well as motivated by the evidence of scalability of reasoning by the CEL reasoner. The standardization of the DL $\mathcal{EL}^{++}$ as the OWL 2 EL potentially attracts new folks of CEL's users. In order to seamlessly integrate the reasoner to the OWL user community, we have implemented the OWL API for CEL. Certain considerations and decisions needed to be made due to the difference in programming languages (between CEL and the OWL API) and efficiency requirements.

This paper describes the challenges, design decision and architecture of our implementation. Additionally, we present experiment results, comparing classification performance of several state-of-the-art DL reasoners. These results not only emphasize the scalability of the CEL reasoner but also demonstrate a reasonably low overhead of our OWL API implementation.

2 OWL 2 EL Profile vs. $\mathcal{EL}^{++}$

Recently, the tractability border of the DLs in the $\mathcal{EL}$ family has been pushed even further [BBL08] to effectively include reflexivity and range restrictions on roles.[3] This gives rise to the new DL $\mathcal{EL}^{++}$ which is more expressive yet tractable. There are also extensions on $\mathcal{EL}^{++}$ that are tractable [KRH08a, KRH08b]. The effort to refine and extend the first Web Ontology Language and to standardize new DLs — among others including $\mathcal{EL}^{++}$— has resulted in a specification proposal of OWL 2 which comprises a number of *profiles*.

Apart from their syntax format and certain regularity conditions on role inclusions (see [HKS06]), the *OWL 2 EL profile* is equivalent to the DL $\mathcal{EL}^{++}$ in terms of logical expressivity. This section recaps the syntax and semantics of the DL and provides correspondence between OWL 2 EL syntactic elements and those of $\mathcal{EL}^{++}$. In DLs, one typically starts with pairwise disjoint sets of *concept names* CN, *role names* RN and *individuals* Ind. Concept and role names in DLs directly correspond to OWL classes and (object) properties, while individuals in DLs are also individuals in OWL's lingo. The upper part of Table 1 summarizes correspondence between $\mathcal{EL}^{++}$ concept constructors and OWL 2 EL class constructors, as well as those functional syntax elements recognized by CEL.

An $\mathcal{EL}^{++}$ ontology (consisting of TBox and ABox components) corresponds to an OWL 2 EL ontology. An ontology is a set of axioms and assertions which are depicted in the lower part of Table 1. Note that a few axiomatic forms are simply syntactic sugar, but they are nevertheless both specified by OWL 2 EL and recognized by CEL for ease of use.

Like most other DLs, the DL $\mathcal{EL}^{++}$— and thus OWL 2 EL profile — employs the set-theoretic semantics. For the semantics as well as its syntactic restriction, we refer the reader to [BBL08, Sun09].

[3] A syntactic restriction limiting the interplay between role inclusions and range restrictions are needed to ensure decidability and tractability of complete reasoning.

DL syntax	CEL native syntax	OWL 2 functional style syntax
$\top$	`top`	Thing
$\bot$	`bottom`	Nothing
$C_1 \sqcap \cdots \sqcap C_n$	`(and` C_1 $\cdots$ C_n`)`	IntersectionOf($C_1 \cdots C_n$)
$\exists r.C$	`(some` r C`)`	SomeValuesFrom(r C)
$C_1 \sqsubseteq C_2$	`(implies` C_1 C_2`)`	SubClassOf(C_1 C_2)
$C_1 \equiv C_2$	`(equivalent` C_1 C_2`)`	EquivalentClasses(C_1 C_2)
$C_1 \sqcap C_2 \sqsubseteq \bot$	`(disjoint` C_1 C_2`)`	DisjointClasses(C_1 C_2)
$r_1 \sqsubseteq r_2$	`(role-inclusion` r_1 r_2`)`	SubPropertyOf(r_1 r_2)
$r_1 \circ \cdots \circ r_n \sqsubseteq s$	`(role-inclusion` `(compose` $r_1 \cdots r_n$`)` s`)`	SubPropertyOf(PropertyChain($r_1 \cdots r_n$) s)
$r_1 \equiv r_2$	`(role-equivalent` r_1 r_2`)`	EquivalentProperties(r_1 r_2)
domain(r) $\sqsubseteq C$	`(domain` r C`)`	PropertyDomain(r C)
range(r) $\sqsubseteq C$	`(range` r C`)`	PropertyRange(r C)
reflexive(r)	`(reflexive` r`)`	ReflexiveProperty(r)
transitive(r)	`(transitive` r`)`	TransitiveProperty(r)
$a_1 = a_2$	`(same-individuals` a_1 a_2`)`	SameIndividual(a_1 a_2)
$a_1 \neq a_2$	`(different-individuals` a_1 a_2`)`	DifferentIndividuals(a_1 a_2)
$C(a)$	`(instance` a C`)`	ClassAssertion(C a)
$r(a_1, a_2)$	`(related` a_1 a_2 r `)`	PropertyAssertion(r a_1 a_2)

Table 1. Correspondence between DL, CEL and OWL 2 EL syntax elements.

3 CEL's Support for Supplemental Reasoning Services

The classical reasoning services supported by most DL systems are all deductive reasoning, where implicit knowledge is deduced from that given explicitly. Two probably most important classical reasoning services at the concept level are subsumption checking (i.e. whether one concept is more general/specific than the other) and satisfiability checking (i.e. whether the concept in question can be interpreted as a non-empty set). At the ontology level, inevitable reasoning services include consistency checking (i.e. whether the ontology has a model) and classification (i.e. computation of the concept hierarchy).

Though necessary, these reasoning services alone are not adequate for realizing a full-fledged ontology development environment. Supplemental reasoning services, such as incremental classification, module extraction and axiom pinpointing are required. Several reasoning techniques for these supplemental services have been proposed in the literature, some of which are generic in the sense that they can be applied to any DLs while some others are logic specific.

Version 1.0 of CEL implements many of these reasoning techniques and natively supports the following reasoning services:

Partial incremental classification Provided that a well-developed ontology $\mathcal{O}$, for instance SNOMED CT, has been classified. Additional axioms $\mathcal{O}^+$ can be asserted to CEL in such a way that CEL reuses the previous classification information together with the new axioms in order to produce classification

results w.r.t. $\mathcal{O} \cup \mathcal{O}^+$. Usage of this reasoning service include incremental development of large ontologies and complex subsumption query answering.

Module extraction Given a large ontology $\mathcal{O}$ and a signature $\mathbf{S}$ (i.e. set of concept and role names) of interest, CEL can efficiently extract a subset $\mathcal{O}_\mathbf{S}$ of $\mathcal{O}$ that preserves the meaning of symbols in $\mathbf{S}$. This supplemental reasoning service has not only been proved helpful in ontology import usage scenarios but can also be exploited as a highly effective optimization of axiom pinpointing.

Axiom pinpointing Given an ontology and a dubious consequence, may it be a subsumption relationship or an unsatisfiable concept, axiom pinpointing can efficiently compute a justification (all justifications) for the consequence. These justifications can then be used to explain or debug that consequence.

For details of the implemented techniques and promising experimental results on several life science ontologies, refer to [Sun09].

4 CEL's Support for the OWL API

The growing use of DL specific tools encouraged us to integrate CEL into the most recent applications. In particular, the OWL API has been successfully used in Protégé[4]. Thus, the OWL API represents a practical tool for connecting with Java, and therefore to the most modern technologies.

One of the main obstacles we found was the disparity in tools for Lisp and Java. We first looked for a tool for connecting these two technologies.

Use of jLinker We found that some tools were available for allowing a connection between Java and Lisp. That is the case of jLinker, a library for Allegro Common Lisp, developed by Franz Inc.[5]. Originally designed for using Java from Lisp, this library did not bring us enough simplicity in its use. In addition, the provided support for data types is insufficient and rather cumbersome to be used. After a few attempts, we considered the primitive but effective way of communication via the operating system: a system call using the standard input and the standard output.

Standard input/output We considered the possibility of using standard input and standard output communication. The purpose was to take advantage of the functionality already provided by the operating system, and the fact that CEL is a compiled program. Even it was theoretically possible to be used in this way, we discarded this architecture after measuring the times needed by the operating system for every system call. Even though we did not rely on the standard input and output for communication, the *system call* idea was still employed in our final architecture.

Since we had discarded the possibility of using Lisp objects from Java, we needed a communication protocol. One of the available standards we had was DIG 1.0.[6]

[4] Protégé is available at: `http://protege.stanford.edu`

[5] `http://www.franz.com/`

[6] `http://dl.kr.org/dig/`

Use of DIG 1.0 CEL supports the DIG 1.1 interface which should make this design option fairly easy to realize. However, the Java part needed a DIG 1.0 writer and parser to communicate with the CEL DIG server. Judging from the potential overhead of conversion between OWL API[7] objects and bulky DIG XML documents as well as from our unsatisfactory experience with overly large communication overhead, we have decided to employ S-expressions instead of DIG.

S-expressions (symbolic expressions) are expressions that represent semi-structured data in human-readable textual form. They consist of symbols and lists, and they are used in Lisp as the representation of source code and data. S-expressions are beneficial in our setting in the sense that they do not require any transformation on the Lisp side, and remain relatively simple for parsing by Java. As a result of our decision, CEL was extended with an interface for following the naming convention of OWL API, and an S-expression parser in Java was developed.

After considering the previous options, we decided to use a system call from Java to start a permanent socket connection, and send and receive S-expressions through the socket. This architecture is explained in detail in the following section. We can see in Table 2 an example of how the OWL API maps the expressivity of OWL 2.

5 The Architecture

In the previous section, we discussed the options we have considered, and those we consider the best ones. In this section, we show how these options were implemented.

We designed a system based on layers. Each layer communicates to the contiguous ones, but cannot use more information from beyond. The mentioned layers are the following: the OWL API interface, the OWL CEL translator, and the connection manager. Besides, there is an independent library, the S-expression parser, which is used by the connection manager. This integration is summarized in the diagram of Figure 1.

The different modules with their function are:

OWL API interface. It follows the OWL API interface implementing an `OWLReasoner`. Its main function is to filter all the unimplemented functions. Only those supported and valid requests are forwarded to the next layer. This layer was especially useful during the development because it can log the functions required by Protégé.

OWL API / CEL translator. This is composed of two different parts:

- a part that translates every OWL API valid object into a CEL S-expression
- a part that translates a CEL S-expression response into an OWL API object

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

OWL 2	OWL API
Class	OWLClass
ObjectProperty	OWLObjectProperty
Individual	OWLIndividual
Thing	OWLThing
Nothing	OWLNothing
IntersectionOf	OWLObjectIntersectionOf
SomeValuesFrom	OWLObjectSomeRestriction
SubClassOf	OWLSubClassAxiom
EquivalentClasses	OWLEquivalentClassesAxiom
DisjointClasses	OWLDisjointClassesAxiom
SubPropertyOf	OWLObjectSubPropertyAxiom
EquivalentProperties	OWLEquivalentObjectPropertiesAxiom
PropertyDomain	OWLObjectPropertyDomainAxiom
PropertyRange	OWLObjectPropertyRangeAxiom
ReflexiveProperty	OWLReflexiveObjectPropertyAxiom
TransitiveProperty	OWLTransitiveObjectPropertyAxiom
SameIndividual	OWLSameIndividualsAxiom
DifferentIndividuals	OWLDifferentIndividualsAxiom
ClassAssertion	OWLClassAssertionAxiom
PropertyAssertion	OWLObjectPropertyAssertionAxiom

Table 2. Mapping between OWL 2 syntactic elements and the OWL API.

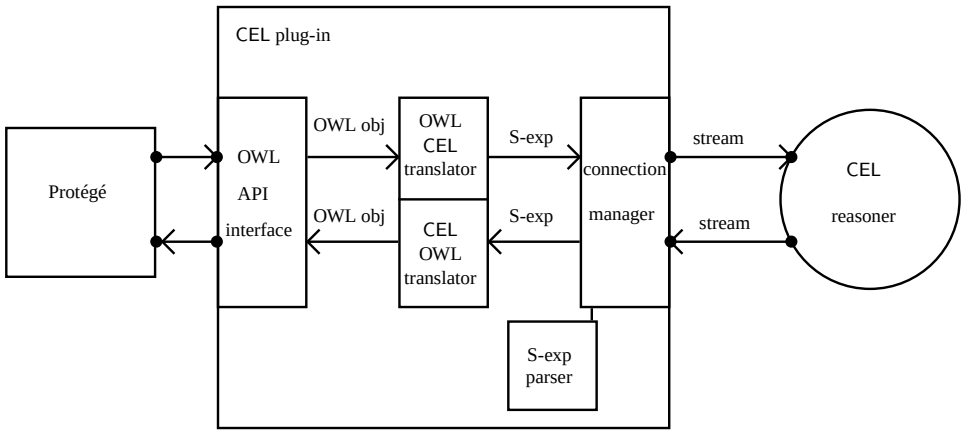

Fig. 1. System architecture.

Besides the translation, this layer manages a small cache in Java. In particular, this layer keeps a reference to the most recent ontology, and a status flag to check whether the ontology has changed. Whenever the ontology is updated, this is cached in this module. In this way, this layer answers queries that can be responded without the need to ask the CEL process.

Connection manager. This layer manages the connection with the CEL reasoner. It keeps track of whether a connection has already been established, and starts a connection when it is necessary. This layer starts the CEL process only with the first request. In case that for some rare circumstances the CEL process is stopped, this layer tries to start a new CEL process. This is technically managed using the connection ports. Port allocation is managed and maintained by the OS, and Java gets a message from the OS whenever a port is occupied. In case that no port is available, an exception is thrown signaling that the CEL process could not be started.

This layer works in the following way:

1. receives a request
2. checks whether there is already an active CEL process
3. if not, copies CEL's executable files into a temporary directory and tries to start a new process
4. converts the S-expression into a stream
5. sends this value to the CEL process, and waits reading the console
6. while it is necessary, updates the progress monitor according to the console
7. parses the stream sent by CEL into an S-expression, and returns this value

From the point of view of the other layers, this one is just a simple Java class. There are not any means to give information about the connection status. Conversely, nothing on this layer is specific to a particular Description Logic. Instead, this layer simply sends and receives S-expressions to and from a process, without interpreting their content.

S-expression parser. For the conversion and parsing of the S-expressions, we used an ad-hoc module. This module is composed of two different parts:

- a part that converts an S-expression into a string
- a part that parses a string to get an S-expression

Although the conversion from S-expressions to strings is quite straightforward, some considerations were taken into account to prevent the overhead of concatenating strings in Java. On the other hand, the parser for S-expressions was designed according to the conventions used in Lisp.

In the following, we present some of the alternatives we discarded.

One of our alternative designs was the use of one single instance of the connection manager for multiple instances of Protégé, and thus ontologies, in the same Java virtual machine. That is a single instance that would keep track of all the connected reasoners. This design was discarded because it was not necessary to have this information in one single object. In particular, socket allocation and management are centralized by the operating system, and the

sockets provided in the operating system provide enough information to open new connections.

We also considered caching the results given by the CEL process. We observed that some queries, for example whether a class is consistent or inconsistent, are frequently performed by Protégé. We had initially thought that caching this sort of information would save communication time. However, we have opted to disable caching because the CEL process already has this information quickly accessible, and the communication time is virtually negligible.

6 Experiments

This section describes experimental results on classification of large-scale biomedical ontologies, comparing several state-of-the-art DL reasoning systems. The second part empirically demonstrate that our architecture and implementation of OWL API is fairly efficient, as classification time with the OWL API overheads is *not* substantially larger than the one in the stand-alone mode.

Ontology test suite

Six realistic large-scale ontologies from the life science domain have been used in our experiments. These are: *the Systematized Nomenclature of Medicine, Clinical Terms* (SNOMED CT), two commonly used versions of *the Galen Medical Knowledge Base (*GALEN*)*, *the Gene Ontology (*GO*)*, *the thesaurus of the US National Cancer Institute (*NCI*)*, and *the Foundational Model of Anatomy (*FMA*)*.[8]

Since GALEN is based on the DL $\mathcal{ELHI}f_{R^+}$ and CEL does not support inverse and functional roles, we actually considered the stripped-down version without these features. Table 3 summarizes the size and other pertinent characteristics of all the test-suite ontologies. The number of axioms is broken down into the following kinds: primitive concept definitions (PCDef), full concept definitions (CDef), general concept inclusions (GCI), role inclusion axioms (RI). The latter also includes domain and range restrictions if present.

Evaluation results

The current version of CEL is written in Common Lisp and compiled and built using Allegro Common Lisp 8.1, and the CEL plug-in for Protégé is written in Java and compiled using Sun's Java Development Kit 1.6.0.[9]

Like most evaluation methods for DL and other reasoning systems, all the experiments described in this section use 'CPU time' as the main performance indicator. Memory consumption is also discussed whenever appropriate. In order to confine the execution environment and hence to induce sensible comparison, the experiments were performed on the same Linux testing server which was

[8] The ontologies are at: `http://lat.inf.tu-dresden.de/~meng/ontologies/`

[9] The tools and sources are available at `http://cel.googlecode.com`.

Ontologies	♯Concepts	♯Roles	♯Axioms			
			PCDef	CDef	GCI	RI
$\mathcal{O}^{\text{Go}}$	20 465	1	19 465	0	0	1
$\mathcal{O}^{\text{Nci}}$	27 652	70	27 635	0	0	140
$\mathcal{O}^{\text{Fma}}$	75 139	2	75 139	0	0	2
$\mathcal{O}^{\text{NotGalen}}$	2 748	413	2 030	695	408	442
$\mathcal{O}^{\text{FullGalen}}$	23 136	950	13 149	9 968	1 951	1 016
$\mathcal{O}^{\text{Snomed}}$	379 691	62	340 972	38 719	0	13

Table 3. The test suite of realistic biomedical ontologies.

equipped with two 2.19 GHz AMD Opteron processors and 2 GB of physical memory.

Since classification is one of the most classical inference services, classification time is often used as a performance indicator for DL systems. A number of state-of-the-art DL reasoners—i.e., FaCT^{++}[10], HermiT[11], KAON2[12] Pellet[13], and RacerPro[14] —were considered for performance comparison. These DL reasoners vary in the sense that they implement different reasoning calculi and are written in different languages. For HermiT, KAON2 and Pellet, Sun's Java Runtime Environment (JRE) version 1.6.0 was used with allotted 1.5 GB heap space. Some reasoners are not equipped with a profiling facility to internally measure CPU time. To achieve comparable measurement, an external timing utility was used with all the classifying systems.

All ontologies in the test suite described in the previous section were used as benchmarks for comparing the performance of the DL reasoners. In the case of $\mathcal{O}^{\text{Snomed}}$, the two complex role inclusions were only passed to CEL and FaCT^{++} but not to the other reasoners, as the latter do not support such axioms. Table 4 shows the (two-run average) time taken by the respective reasoners to classify the biomedical ontologies, where *m/o* means that the reasoner failed due to memory exhaustion, and *t/o* means that the reasoner did not terminate within the allocated time of 24 hours. Figure 2 depicts a comparison chart of reasoners' performance based on their classification time, where *m/o* and *t/o* are displayed as full vertical bars.

It can be seen from the chart and the table that CEL is the only DL reasoner that can classify all six biomedical ontologies in the test suite and outperforms Pellet, HermiT and KAON2 in all cases. Compared with the other reasoners, CEL is faster than FaCT^{++} and RacerPro w.r.t. all but $\mathcal{O}^{\text{Nci}}$ and $\mathcal{O}^{\text{Snomed}}$. It should be noted that, when it first came into existence in 2005 [BLS05], CEL was the only

[10] `http://owl.man.ac.uk/factplusplus/`
[11] `http://www.hermit-reasoner.com`
[12] `http://kaon2.semanticweb.org`
[13] `http://clarkparsia.com/pellet/`
[14] `http://www.racer-systems.com`

Ontologies	$\mathcal{O}^{\text{GO}}$	$\mathcal{O}^{\text{NCI}}$	$\mathcal{O}^{\text{FMA}}$	$\mathcal{O}^{\text{NOTGALEN}}$	$\mathcal{O}^{\text{FULLGALEN}}$	$\mathcal{O}^{\text{SNOMED}}$
CEL	0.98	3.75	9.04	2.83	201	1 258
FaCT^{++}	20.12	1.72	*t/o*	3.28	*m/o*	606
HermiT	16.75	34.92	123	12.35	*m/o*	*m/o*
KAON2	*m/o*	*m/o*	*t/o*	*m/o*	*m/o*	*t/o*
Pellet	52.58	36.11	7 753	31.56	*m/o*	*m/o*
RacerPro	17.11	13.36	629	17.06	*t/o*	1 155

Table 4. Computation time (second).

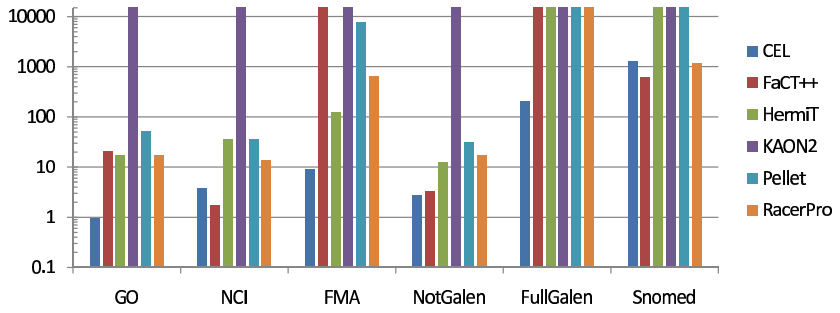

Fig. 2. Performance comparison through classification time (second).

academic DL system that was capable of classifying entire SNOMED CT. This has subsequently sparked interest in the DL community to research on optimization techniques specific to the biomedical ontologies (in particular, to SNOMED CT), and later enabled tableau-based reasoners like FaCT^{++} to take advantage of simple structures of ontologies of this kind. These reasoners employed some of the optimization techniques described in [HT05, HMW08] that are highly effective on simpler TBoxes (i.e., without GCIs) like $\mathcal{O}^{\text{SNOMED}}$. However, when a large number of GCIs are present as in the case of $\mathcal{O}^{\text{FULLGALEN}}$, these reasoners fail due to either memory exhaustion or time out. Interestingly, CEL is the only reasoner that can classify $\mathcal{O}^{\text{FULLGALEN}}$.

According to our preliminary experiments on OWL API, classification time by the CEL reasoner through OWL API and Protégé are not much larger than those by the *stand-alone* CEL reasoner. Considerably more memory consumption by the former setting was expected since Protégé and our OWL API implementation need to maintain their own data structures. In particular, Protégé has a representation of the entire ontology in form of OWL API objects.

7 Concluding Remarks

In this paper, we have described a new extension to the CEL reasoner, an OWL API implementation and reasoner plug-in for Protégé, which shows CEL's rea-

soning capabilities to Protégé users. We presented the challenges, design decision and architecture of this implementation.

The CEL system is known for its scalability of reasoning in DL $\mathcal{EL}^{++}$. As shown in the paper, this logic is suitable for several ontology applications, most notably from the life science domain. The presented extension allows the connection between the OWL API and CEL. In addition, we presented results that remark the scalability of the reasoner, with a reasonably low overhead of our OWL API implementation. Since DL $\mathcal{EL}^{++}$ has been taken as the basis for the OWL 2 EL profile of the new Web Ontology Language, new users might find CEL a useful tool. We have implemented the OWL API for CEL to seamlessly integrate the reasoner to the OWL user community.

References

[BBL08] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the $\mathcal{EL}$ envelope further. In Kendall Clark and Peter F. Patel-Schneider, editors, *In Proceedings of the OWLED 2008 DC Workshop on OWL: Experiences and Directions*, 2008.

[BLS05] Franz Baader, Carsten Lutz, and Boontawee Suntisrivaraporn. Is tractable reasoning in extensions of the description logic $\mathcal{EL}$ useful in practice? In *Proceedings of the 2005 International Workshop on Methods for Modalities (M4M-05)*, 2005.

[HKS06] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible $\mathcal{SROIQ}$. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*, pages 57–67. AAAI Press, 2006.

[HMW08] V. Haarslev, R. Möller, and S. Wandelt. The revival of structural subsumption in tableau-based description logic reasoners. In *Proceedings of the 2008 International Workshop on Description Logics (DL2008)*, CEUR-WS, 2008.

[HT05] I. Horrocks and D. Tsarkov. Optimised classification for taxonomic knowledge bases. In *Proceedings of the 2005 International Workshop on Description Logics (DL'05)*, pages 184–191, 2005.

[KRH08a] Markus Krötzsch, Sebastian Rudolph, and Pascal Hitzler. Description logic rules. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *Proceedings of the 18th European Conference on Artificial Intelligence*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 80–84. IOS Press, 2008.

[KRH08b] Markus Krötzsch, Sebastian Rudolph, and Pascal Hitzler. Elp: Tractable rules for owl 2. pages 649–664. 2008.

[Law08] Michael Lawley. Exploiting fast classification of SNOMED CT for query and integration of health data. In *Proceedings of the 3rd Knowledge Representation in Medicine Conference (KR-MED'08): Representing and Sharing Knowledge Using SNOMED*, Phoenix AZ, USA, 2008.

[Sun09] Boontawee Suntisrivaraporn. *Polynomial-Time Reasoning Support for Design and Maintenance of Large-Scale Biomedical Ontologies*. PhD thesis, TU Dresden, Institute for Theoretical Computer Science, Germany, 2009.