jcel: A Modular Rule-based Reasoner

Julian Mendez

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

Abstract. jcel is a reasoner for the description logic $\mathcal{EL}^+$ that uses a rule-based completion algorithm. These algorithms are used to get subsumptions in the lightweight description logic $\mathcal{EL}$ and its extensions. One of these extensions is $\mathcal{EL}^+$, a description logic with restricted expressivity, but used in formal representation of biomedical ontologies. These ontologies can be encoded using the Web Ontology Language (OWL), and through the OWL API, edited using the popular ontology editor Protégé. jcel implements a subset of the OWL 2 EL profile, and can be used as a Java library or as a Protégé plug-in. This system description presents the architecture and main features of jcel, and reports some of the challenges and limitations faced in its development.

1 Introduction

This system description presents jcel[1], a reasoner for the description logic $\mathcal{EL}^+$. The design and implementation details refer to jcel 0.17.1, unless other version is specified.

The lightweight description logic (DL) $\mathcal{EL}$ and its extensions have recently drawn considerable attention since important inference problems, such as the subsumption problem, are polynomial in $\mathcal{EL}$ [1,4,2]. In addition, biomedical ontologies such as the large ontology SNOMED CT[2], can be defined using this logic.

The basic entities are *concepts* (class expressions), which are built with *concept names* (classes) and *role names* (object properties).

An *ontology* is a formal vocabulary of terms which refers to a conceptual schema inside a domain. The terms are related using an *ontology language.* The main service that this reasoner provides is *classification*, a hierarchical relation of the concepts in the ontology.

2 Language Subset Supported

jcel, as an $\mathcal{EL}^+$ reasoner, includes the standard constructors of $\mathcal{EL}$: conjunction ($C \sqcap D$), existential restriction ($\exists r.C$), and the top concept ($\top$). In addition, this logic includes role composition ($r \circ s \sqsubseteq t$) and role hierarchy ($r \sqsubseteq s$).

[1] http://jcel.sourceforge.net/
[2] http://www.ihtsdo.org/snomed-ct/

Table 1. Syntax and semantics.

Name	Syntax	Semantics
concept name	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
role name	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
top	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
conjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
existential restriction	$\exists r.C$	$(\exists r.C)^{\mathcal{I}} = \{x \mid \exists y : (x, y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
subsumption	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
role hierarchy	$r \sqsubseteq s$	$r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$
role composition	$r \circ s \sqsubseteq t$	$r^{\mathcal{I}} \circ s^{\mathcal{I}} \subseteq t^{\mathcal{I}}$

Table 1 summarizes the semantics of the mentioned constructors.

There is an experimental development of jcel [7] to support inverse and functional roles.

3 Syntaxes and Interfaces Supported

jcel is developed in Java and can be compiled in Java 1.6 or Java 1.7 indistinctly. jcel is integrated to the OWL API 3.2.4[3], which is a Java application programming interface (API) and reference implementation for creating, manipulating and serializing OWL ontologies. Using the OWL API, jcel can be used as a Protégé[4] plug-in. Protégé is a free, open source ontology editor, and also a knowledge base framework. Protégé ontologies can be exported to several formats, like RDF, OWL and XML Schema. In order to keep compatibility with Protégé 4.1, jcel is distributed as a Java binary for Java 1.6.

jcel can also be used as a library without the OWL API. It has its own factories to construct the optimized data types used in the core.

4 Reasoning Algorithm Implemented

The algorithm is rule-based, and there is a set of rules that are successively applied to saturate data structures. These rules are called *completion rules*. The process is called *classification*, and is the main part of jcel's algorithm.

An algorithm that classifies the set of axioms by applying these rules could be expensive in time if it is performed by a systematic search. The algorithm used by jcel is based on CEL's algorithm [2,3], but generalized with a *change propagation* approach [6]. This approach detects the changes in the data structure being saturated, and triggers the rules in consequence.

The input of the algorithm is a *normalized* set of axioms $\mathcal{T}$ as in the following list: $A \sqsubseteq B$, $A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B$, $A \sqsubseteq \exists r.B$, $\exists r.A \sqsubseteq B$, $r \sqsubseteq s$, $r \circ s \sqsubseteq t$.

[3] http://owlapi.sourceforge.net/

[4] http://protege.stanford.edu/

The invariant of the algorithm has a set S, called *set of subsumptions*, such that for each pair of concept names A, B in $\mathcal{T}$: $(A, B) \in S$ if and only if $\mathcal{T} \models A \sqsubseteq B$, and a set R such that for each triple of role r, and concept names A, B in $\mathcal{T}$: $(r, A, B) \in R$ if and only if $\mathcal{T} \models A \sqsubseteq \exists r.B$, where $\models$ has the usual meaning.

The algorithm finishes when S is saturated. The output is S itself, which tells the subsumption relation for every pair of concept names.

In Figure 1, we can observe S and R, the completion rules (CR-1, CR-2, ...), the duplicates checker, and a set Q which has a set of entries to be processed.

In each iteration, an S-entry (a pair) or an R-entry (a triple) is taken from Q and added to either S or R. This change is propagated and sent to the chain of rules sensitive to changes in S (S-chain) or in R (R-chain). Every completion rule takes the new entry as input and returns a set of S- and R-entries. The arrows indicate how these entries flow.

Every element proposed by the rules is verified by the duplicates checker before being added to Q. The dashed lines indicate that the duplicates checker uses S and R for the verification. This procedure is repeated until Q is empty.

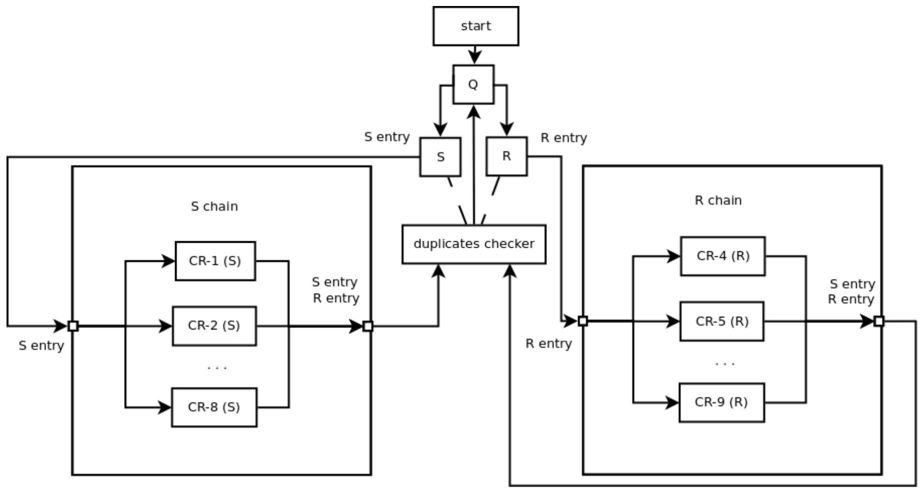

Fig. 1. Dynamic diagram.

5 Architecture and Optimization Techniques

jcel axioms are encoded using integers. This optimizes the use of memory and required time in comparisons. Any program using jcel as a library can encode its entities with integers, and take advantage of this efficient representation.

jcel is composed by several modules, as shown in Figure 2. The arrows indicate the relation “depends on”.

jcel.coreontology and jcel.core are modules that use only normalized axioms. The former contains the normalized axioms themselves, the latter contains the implementation of the completion rules, together with the data structures for the subsumption graphs.

jcel.ontology is a module that contains the axioms for the ontology and the procedures to normalize the ontology. jcel.reasoner is the reasoner interface. It can classify an ontology and compute entailment.

All the modules mentioned above work with data types based on integers.

jcel.owlapi is the module that connects to the OWL API. This module performs the translation between the OWL API axioms and jcel axioms. jcel.protege is a tiny module used to connect to Protégé.

Figure 2 describes the module dependencies and shows that jcel can be used:

- extending the core (with jcel.coreontology and jcel.core)
- as a library using integers (adding jcel.ontology and jcel.reasoner)
- as a library using the OWL API (adding jcel.owlapi)
- as a Protégé plug-in (adding jcel.protege)

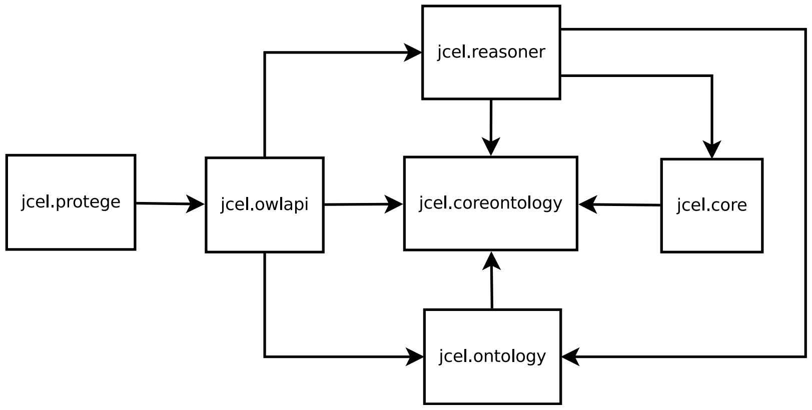

Fig. 2. Module dependencies.

6 Particular Advantages and Disadvantages

jcel is a **pure Java**, **open source** project. Its source code can be cloned with Git[5] and compiled using Apache Ant[6] or Apache Maven[7] indistinctly.

[5] http://git-scm.com/

[6] http://ant.apache.org/

[7] http://maven.apache.org/

jcel includes several advantages derived from its design and from good practices of software engineering. Regarding the design, jcel has **independent maintenance of rules**. Each rule works as an independent unit that can be implemented, improved and profiled independently.

Regarding the good practices, jcel uses **no null pointers**. Every public method is prevented of accepting null pointers (throwing a runtime exception) and no public method returns a null pointer. Referred by its inventor as "The Billion Dollar Mistake"[8], a null pointer may have different *intended meanings*. For example, $min(1, \mathbf{null})$, may give the results "0" (considering **null** as 0), "1" (considering **null** as an empty argument), or "**null**" (considering **null** as an undefined value).

Another good practice is that public methods return **unmodifiable collections** when they refer to collections used in the internal representation. This prevents a defective piece of code from modifying the collection.

jcel has **no cyclic dependencies of packages** in each module. This facilitates maintenance, since modifications on one package do not alter any other package that does not depend on the former.

jcel has also some points that are not implemented yet, but can be considered as good future improvements.

One of the improvements is to apply techniques to **unchain properties** [5]. This would be useful for large ontologies.

Another improvement is **incremental classification**. This could be especially interesting for entailment, since jcel computes entailment by adding fresh auxiliary concepts and reclassifying the ontology.

Finally, the **reduction of use of memory** is an important improvement to consider. jcel 0.16.0 was the first version to include entailment, but also became significantly slower than jcel 0.15.0 when classifying large ontologies. The reason was a side effect resulting from the memory required for the entailment, producing a frequent execution of the garbage collector. This was partially solved in jcel 0.17.0.

7 Application focus

jcel can be used to classify small and medium-sized ontologies of the $\mathcal{EL}$ family.

jcel was designed to be robust, resilient, modular and extensible. Its binaries are small and can be distributed inside other tools. One tool that uses jcel is Gel[9] (Generalizations for $\mathcal{EL}$) [7], which extends the core of jcel.

Another example is OntoComP[10]. It is a Protégé plug-in for completing OWL ontologies, and for checking whether an OWL ontology contains "all relevant information" about the application domain. This tool uses the OWL API to connect to jcel as a library.

[8] http://qconlondon.com/london-2009/speaker/Tony+Hoare

[9] http://sourceforge.net/p/gen-el/

[10] http://ontocomp.googlecode.com/

jcel is also used to verify correctness in UEL[11], Unification for the description logic $\mathcal{EL}$.

8 Conclusion

jcel is a reasoner for lightweight ontologies. Its rule-based design makes it easy to be configured according to the rules. Provides a high-level interface to be used as a tool seamlessly integrated to the OWL API.

The implementation is modular, resilient and highly extensible. Implemented in a state-of-the-art technology, it has a low coupling and a high cohesion, it is portable, and has an optimal interface to connect to other technologies of the Semantic Web.

References

1. Franz Baader. Terminological cycles in a description logic with existential restrictions. In *Proc. IJCAI'03*, 2003.
2. Franz Baader and Sebastian Brandt and Carsten Lutz. Pushing the $\mathcal{EL}$ envelope. In *Proc. IJCAI'05*, 2005.
3. Franz Baader and Carsten Lutz and Boontawee Suntisrivaraporn. Is Tractable Reasoning in Extensions of the Description Logic $\mathcal{EL}$ Useful in Practice?. Journal of Logic, Language and Information, Special Issue on Method for Modality (M4M), 2007.
4. Sebastian Brandt. Polynomial time reasoning in a description logic with existential restrictions, GCI axioms, and—what else? In *Proc. ECAI'04*, 2004.
5. Yevgeny Kazakov and Markus Krötzsch and František Simančík. Unchain My EL Reasoner. In Riccardo Rosati, Sebastian Rudolph, Michael Zakharyaschev, editors, *Proceedings of the 24th International Workshop on Description Logics (DL-11)*. CEUR, 2011.
6. Julian Mendez. A Classification Algorithm for $\mathcal{ELHI}fR^+$. Dresden University of Technology, 2011.
7. Julian Mendez and Andreas Ecke and Anni-Yasmin Turhan. Implementing completion-based inferences for the $\mathcal{EL}$-family. In Riccardo Rosati, Sebastian Rudolph, and Michael Zakharyaschev, editors, *Proceedings of the international Description Logics workshop*, volume 745. CEUR, 2011.
8. Quoc Huy Vu. Subsumption in the Description Logic $\mathcal{ELHI}fR^+$ w.r.t. General TBoxes. Dresden University of Technology, 2008.

[11] http://uel.sourceforge.net/