DLRule: A Rule Editor plug-in for Protégé

Francis Gasse and Volker Haarslev

Concordia University, Montreal, Quebec, Canada
{ f_gasse,haarslev } @cse.concordia.ca

Abstract. OWL is a very expressive language, but some user obviously struggle to formulate what they want to say. Now, some of these users may find it easier to write down a SWRL rule instead of an OWL axiom. Hence, we present a rule editor plug-in for Protégé that brings something different to rule and OWL integration. We part from the two usual approaches: (i) use it as is with say, Hoolet, knowing that it leads to undecidability. (ii) Or make it DL-safe, but then it restricts the semantic impact and, e.g., looses the nice "car owners are engine owners" inference. This plug-in implements a rewriting technique that rewrites certain forms of rules into DL axioms using OWL 1.1's new features. These rules rewritten as OWL 1.1 axioms do not require DL-safety, thus allow the extra inferences, and do not cause any undecidability. In this paper, we outline the rewriting technique, present the plug-in and give some practical results.

1 Introduction

Ontologies are used more and more for different applications in a lot of domains, notably the Semantic Web and the life sciences. These ontology are often written in OWL, which uses Description Logics (DL) [1] as logical foundation. The forms of knowledge that has to be represented across these usages are not always trivial to encode in OWL. A notable example of such problematic type of knowledge is rule-like knowledge. Rules are very natural and intuitive to model some domains. The problem is that reasoning over an OWL ontology augmented with (SWRL [2]) rules is undecidable. We can regain decidability by restricting rules to be DL-safe [3]. DL-safety means that only individuals explicitly introduced in the ontology can be matched against the variables of the rule. The main drawback of this restriction is that it restricts the semantic impact of the rule, e.g., it prevents the nice "car owners are engine owners" type of inference, where one can derive that the owner of a car also owns the engine of that car. Another option was recently introduced in [4]. This technique identifies a certain form of rules that can be rewritten into OWL axioms using features introduced in OWL 1.1 [5]. Since the rules are encoded as axioms, they are not restricted to DL-safety thus the aforementioned inference is preserved. This could allow a user more used to writing rules to define a complete ontology as a set of rules and it would have the same semantics as its OWL counterpart. Another benefit of this rewriting is that we can still use standard OWL reasoners. This rewriting obviously does not

add expressivity to OWL 1.1, but it uses the available constructs in a non-trivial way to maximize the expressive power of the rules. Here are a few examples of the rules this rewriting can handle: (i) $R(x,y) \wedge R(x,z) \wedge C(y) \wedge D(z) \rightarrow C(x)$, (ii) $R(x,y) \wedge S(y,z) \rightarrow S(x,y)$, (iii) $\mathit{hasOffspring}(y,x) \wedge \mathit{hasParent}(x,y) \wedge \mathit{hasSibling}(y,z) \wedge \mathit{Man}(z) \rightarrow \mathit{hasUncle}(x,z)$. The rule (i) can be rewritten as a standard concept definition and is thus obviously rewritable. The rules (ii) and (iii) are much more interesting because they are not obvious to rewrite, even for an experienced OWL user.

In this paper we present a plug-in for the Protégé[1] ontology editor, the DL-Rule plug-in[2]. This plug-in is a feature-rich rule editor that allows the user to create and edit rules within Protégé. The defining characteristic of this plug-in is that it implements the rewriting introduced earlier so that if a rule is rewritable, it will be added to the ontology as axioms to get the most inference possible and if it is not rewritable, the rule is then added to the ontology as a SWRL rule to be handled by a reasoner supporting DL-safe rules. By using these two approaches in conjunction, we get the "best of both world", so to speak, since each rule is added to the ontology in the form that suits it best. The structure of the paper is as follows. In Section 2 we introduce some notions of OWL 1.1 needed to understand this paper. We introduce the forms of rules that can be rewritten in Section 3, and we give an overview of Protégé in Section 4. We then describe the plug-in itself and its features in Section 5 and we finish by rewriting a rule to give some intuition of how the rewriting works in Section 6.

2 OWL 1.1

The rewriting technique we implemented in the plug-in requires features that are part of the OWL 1.1 submission. OWL 1.1 extends OWL-DL with many new features but here, due to space constraints, we will focus on the two new features that are used extensively in the rewriting: the self restriction and the sub-property chain. The self restriction construct models individuals that are connected to themselves by a given object property. The classical example to illustrate the utility of this to define the class Narcissist with a self restriction on the property likes. The sub-property chain feature expresses the propagation of an object property over the composition of a sequence of object properties. For instance, on could model the propagation of the hasLocation property over the partOf property, meaning that if x is a part of y and y has location z then x has location z. However, the form of the chain and the property hierarchy as a whole is subject to some restrictions to ensure decidability. We will now explain these restrictions.

We first need to define a relation, $\sqsubseteq$, as the smallest relation on properties for which the following holds. If the ontology contains an axiom of the form

– SubObjectPropertyOf(P1 P2), then P1 $\sqsubseteq$ P2 holds;

[1] http://protege.stanford.edu
[2] http://users.encs.concordia.ca/~f_gasse/

- EquivalentObjectProperties(P1 P2), then P1 $\sqsubseteq$ P2 and P2 $\sqsubseteq$ P1 hold;
- InverseObjectProperties(P1 P2), then P1 $\sqsubseteq$ INV(P2) and INV(P2) $\sqsubseteq$ P1 hold;
- SymmetricObjectProperty(P), then P $\sqsubseteq$ INV(P) holds;
- P1 $\sqsubseteq$ P2 holds, then INV(P1) $\sqsubseteq$ INV(P2) holds as well.

The relation $\sqsubseteq^*$ is the reflexive-transitive closure of $\sqsubseteq$. An object property P is composite if it, or its inverse, is either the super-property of a property chain, or transitive. An object property P is simple if, for each object property P' such that P' $\sqsubseteq^*$ P, P' is not composite. An ontology is valid in OWL 1.1 if it satisfies the following conditions:

- Cardinality restriction and self restriction axioms only contain simple properties.
- Only simple properties are defined as functional, inverse functional, irreflexive, asymmetric or disjoint.
- There must exist a strict partial ordering $\prec$ for which the following holds:
  - If x $\prec$ y holds, then y $\sqsubseteq^*$ x does not hold;
  - Each axiom of the form SubObjectPropertyOf(SUB P) where SUB is of the form SubObjectPropertyChain($P_1 ... P_n$) with n $\geq$ 2 fulfills the following conditions:
    - n = 2 and $P_1 = P_2 = P$, or
    - $P_i \prec P$ for each $1 \leq i \leq n$, or
    - $P_1 = P$ and $P_i \prec P$ for each $2 \leq i \leq n$, or
    - $P_n = P$ and $P_i \prec P$ for each $1 \leq i \leq$ n-1.

These restrictions prevents the rewriting of some common forms of rules. For example, it does not allow the specialization of a property w.r.t. its filler such as the rule $hasChild(x, y) \wedge Man(y) \rightarrow hasSon(x, y)$ would induce. Indeed, from the rule we have $hasChild \prec hasSon$ and since *hasSon* is a subproperty of *hasChild*, $hasSon \sqsubseteq^* hasChild$ also holds and this violates the definition of $\prec$.

3 Admissible rules

We will now define what form of rules are admissible to be rewritten, but first we have to define a rule. Let X be a set of variables disjoint from role or concept names. An *atom* is either a class atom $C(x)$ or a property atom $R(x, y)$, for C a class, R a property and x, y variables from X. A *rule* is an expression of the form $A_1 \wedge A_2 \wedge ... \wedge A_n \rightarrow H$ where A_i and H are atoms of either form, and the variables in H occur in some of the A_i. We will use a graph conceptualization of the rules to define the admissible rules. We do so because it allows us to state restrictions in an elegant way and it is a is rather intuitive way to see rules. We now introduce the relevant graph related notions.

A *rule graph* is a directed labeled graph induced by the body of the rule. Given a rule, its graph is defined as follows. For every atom of the form $C(x)$, there is a node x and its label contains C. For every atom of the form $R(x, y)$, x and y are nodes and there is an edge $\langle x, y \rangle$ labeled R. Nominals in atoms are

treated the same as variables, but the nominal has to be added to the label of the node. For example, for an atom $R(Nominal, x)$ there is a node *nominal*, labeled $Nominal$, and an edge $\langle nominal, x\rangle$ labeled R. To improve readability, when we say that a graph contains an edge $\langle x, y\rangle$, labeled R, it is implied that it could instead contain $\langle y, x\rangle$, labeled R^-. We now introduce the notion of a skeleton of a rule graph.

A graph is a skeleton, with respect to a property hierarchy, if there are no edges that are implied by other edges of the graph. An edge can be implied by edges of a sub-property, an inverse, a subproperty chain or itself, if it is transitive. A skeleton of a rule graph is obtained by applying the following rules until none is applicable anymore:

1. Let S be the inverse property of R. If $\langle x, y\rangle$ labeled R and $\langle y, x\rangle$ labeled S are in the graph, then remove one of them.
2. Let R be a transitive property. If $\langle x, y\rangle, \langle y, z\rangle$ and $\langle x, z\rangle$ all labeled R, are in the graph, then remove the $\langle x, z\rangle$ edge.
3. Let R be a super-property of S. If $\langle x, y\rangle$ labeled R, and $\langle x, y\rangle$ labeled S and $R \neq S$, then remove the R edge.
4. Let a subproperty chain of the form $R_1, R_2, ..., R_n$ be the subproperty of S. If $\langle x_1, x_2\rangle$ labeled R_1, $\ldots$, $\langle x_n, x_{n+1}\rangle$ labeled R_n, and $\langle x_1, x_{n+1}\rangle$ labeled S are in the graph, then remove the S edge.

The reason we use skeletons is that by removing implied edges from the rule graph, we maximize the chances that it satisfies the restrictions imposed on the form of the rules. There are two kind of rules, concept-headed and property-headed. They have different admissibility conditions, so we will define both separately. A rule graph G is *concept-headed rule admissible* if it is a tree, i.e., does not contain any cycle. To define the restrictions over the property-headed rules, we first have to introduce the notion of the *main chain* of a rule's skeleton graph, which is the path between the two nodes referenced in the rule's head, including these. A rule graph G is *property-headed rule admissible* if:

1. It is a tree.
2. Let l be the list of the labels of the edges of the main chain and P be the property in the head of the rule
   (a) l is of one of the valid form for property chain listed in Section 2, and
   (b) if P is the first (resp. last) property in l, then the first (resp. last) node does not contain class assertions and P is the only edge connected to it.

These restrictions are necessary since we rewrite the main chain as a RIA. We have to add further restriction if P is the first or last role in l because the presence of class assertions or other edges in the theses nodes would cause the insertion of other properties at either end of l during the rewriting. In such a case, P would not be the first (last) role in l and the resulting RIA would be invalid.

4 Protégé

Protégé is an open-source, Java-based ontology development environment that is developed by Stanford University and University of Manchester. In its basic form, Protégé can edit OWL ontologies graphically, browse the class and property hierarchies, etc. The reasoning is handled by third-party reasoners. Two reasoners, FaCT++ [6] and Pellet [7], are bundled with Protégé and any DIG enabled reasoner can also be used. Protégé's architecture was created with extensibility in mind, so a framework was provided to enable the development of plug-ins by third party developers. There exists a lot of plug-ins[3] that offer a variety of services, from visualization to conversion to other formalism. This extensibility and customizability make Protégé extremely versatile. Protégé's extensibility and its large user base were the main motivations to implement the rewriting within that framework.

The current official release of Protégé (3.3.1) does not offer support for OWL 1.1, which we need, so we have to use an alpha release (4.0) that is however quite stable. Obviously, the reasoner used has to support OWL 1.1 and the reasoners bundled with Protégé do so.

5 DLRule plug-in

We now describe the plug-in and the way it works. We present the different components composing it, how it is integrated within Protégé and how to edit rules using it. The plug-in appears as a tab within the Protégé interface labeled "DLRules". There are six main components in the tab (the numbers refer to Figure 1):

1. The class pane is a standard Protégé component which displays the class hierarchy of the current ontology as a tree.
2. The property pane is a standard Protégé component which displays the property hierarchy of the current ontology as a tree.
3. The rule list contains all the rules in the project. The selected rule is displayed in the editor.
4. The text rule editor is where the user types in the rules (syntax is defined in Section 5.1). Properties and classes can be dragged from the panes and dropped to create an atom.
5. The graphical rule viewer shows a graphical representation of the rule currently edited. This allows the user to validate the rule before inserting it into the ontology in a more intuitive way than textually.
6. The component labeled "Actions" is where the user can launch the different tasks executed by the plug-in. These tasks are explained in Section 5.3
7. The rule previewer lets the user see what effects the rules will have on the class hierarchy.

[3] `http://protege.stanford.edu/plugins/`

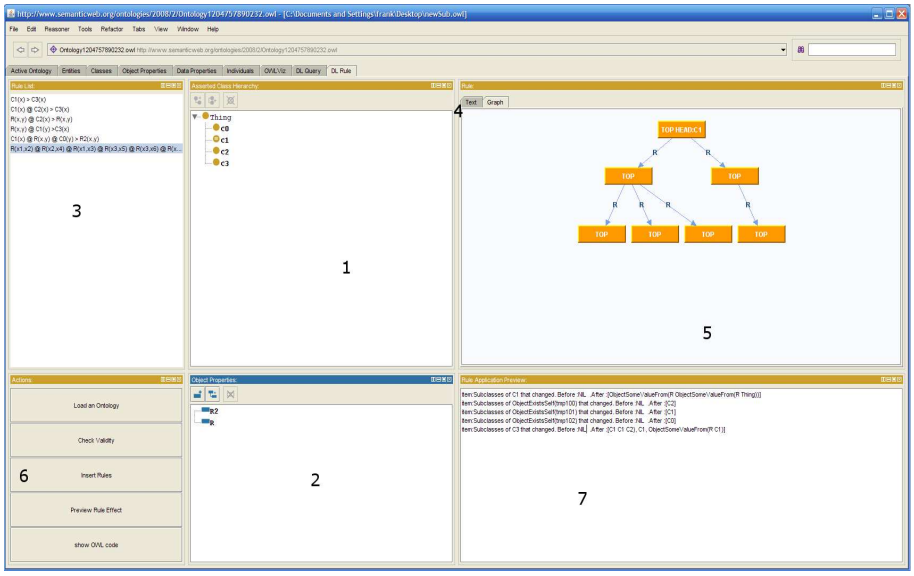

Fig. 1. Overview of the DLRule plug-in.

5.1 Rule Syntax

The syntax of a rule we use in the editor is really close to the usual abstract syntax of rules. The syntax of the rules is defined in the following grammar, written using EBNF notation:

```
<rule> = <atom> [ ,"@" <atom>  ] , ">" , <head>;
<atom> = <propertyAtom> | <classAtom> ;
<head> = <propertyAtom> | <classAtom>;
<propertyAtom> = <propertyName>,"(",<variable>,",",<variable>,")";
<classAtom> = <className>,"(",<variable>,")";
```

In the above grammar, `<className>` and `<propertyName>` refer to classes and properties present in the ontology and `<variable>` is a string starting with a letter and composed only of letters and digits.

5.2 Ontology I/O

The quantity of axioms, classes and properties that are added to an ontology by the rewriting can be rather large, so browsing and editing the ontology can get confusing for the user since he has to discriminate the domain axioms from the axioms generated by the rewriting. Thus, we provided a mechanism to allow the user to browse and edit an ontology containing rules without seeing the properties and classes created by the rewriting. We developed a module that

handles the input/output tasks for ontologies containing rules. The general idea of this module is that it isolates the rules from the other tabs of Protégé. The loading and saving of rule augmented ontologies has to be launched from the DLRules tab, not from Protégé's menu, for it to use this code.

Save To save an ontology and rules, for all rule we verify if it is rewritable. If so, we add the necessary OWL axioms to the ontology, otherwise we build a SWRL rule and add it to the ontology. The ontology is then actually saved.

Load When loading an ontology containing rules, the file is traversed and all the axioms marked as generated by the rewriting component are removed from the ontology and set aside. We then load this "cleaned" ontology into Protégé and reverse engineer the rules from the set of axioms we isolated earlier. These rules are then loaded into the tab.

The isolation between Protégé and the rules we mentioned above has a side-effect. Since Protégé loads a cleaned version of the ontology, it is not aware of the changes to the class hierarchy caused by the rules, such as a new subsumption. That can lead to all sorts of problems, for example a rule could cause a class to be unsatisfiable and it would not show in Protégé (except in the DLRule tab). In order to overcome this problem, when we create the clean ontology we add axioms to assert explicitly all the rule induced changes in the classes hierarchy. This way, we maintain the separation between Protégé and the rules while ensuring that Protégé works with the valid class hierarchy.

5.3 How to use the plug-in

We tried to maximize the plug-in's usability while designing it and it resulted in a plug-in that is rather intuitive and simple to use. Even so, we will outline how to carry on the most common tasks. The "Text" tab is where the user can edit the rules in a text-based mode. The user can type in the rule and can also drag a property or a class from the arborescences over to where in the rule the atom is going and drop it. This will insert the structure of an atom in the rule, leaving only the variables left to type. The "Graph" tab shows a visual representation of the rule, in which the rule is shown as a graph. The rule's variables are nodes in the graph, the property atoms are edges and class atoms are the labels of the node. Such a graphical representation is very intuitive and allows the user to easily validate the rule.

The Actions zone This area contains the components that launch all the tasks carried on by the plug-in. We will now describe each of these tasks, how to use them, when to use them and their benefits.

Check Validity The conditions that have to be met for a rule to be admissible to the rewriting are not always so easy to verify and the reasoners do not necessarily warn the user that the resulting ontology is not valid w.r.t. OWL 1.1's

specification. This feature verifies that the rules are all admissible to be rewritten and that the property hierarchy of the rewritten ontology is valid, w.r.t. OWL's restrictions. The errors, if any, will be displayed so that the user can correct them. This ensures that the reasoning process will be sound, which may not be the case with a flawed ontology.

Simulate Rule Application Rules are a powerful mechanism and can affect the ontology in many ways that a user cannot always easily foresee. To help the user to ensure that the addition of a rule will not have unwanted effects, we provide a simulator that will list all the changes that the re of the rule would imply.

Load an ontology As we mentioned earlier, we override Protégé's I/O system so we can hide the axioms generated by the rules that are irrelevant to the user. To enjoy these benefits, the user has to launch the load process from here.

Insert Rules When a user creates a rule, it is not rewritten and added to the ontology right away. It is that way to prevent the insertion of a rule that would not be completed. The insertion of the rules in the ontology has to be launched explicitly by the user using this component. The ontology is automatically saved after the rules are inserted and a cleaned version of the ontology is reloaded in Protégé afterward.

6 Example

In order to illustrate how the rewriting works, we will now rewrite a rule step by step. The rewriting of class-headed rules is quite intuitive so we will rewrite a property-headed rule. Since the "uncle" example was used so often to demonstrate the motivations to have rules integrated with OWL, we will use it for the example. Obviously, we can write a subproperty chain axiom to model this problem in OWL 1.1, e.g., SubObjectPropertyOf(SubObjectPropertyChain (hasParent, hasBrother) hasUncle), but we will assume that we do not have hasBrother to show the potential of the rewriting. Let us assume that the rule is formulated as follows: $hasOffspring(y, x) \wedge hasParent(x, y) \wedge hasSibling(y, z) \wedge Man(z) \rightarrow hasUncle(x, z)$. The rule graph of this rule is shown in Figure 2.

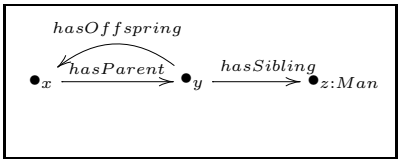

Fig. 2. The rule graph of the example

The first step is to make this rule graph a skeleton by removing implied edges. In our example ontology, we have SubObjectPropertyOf (hasParent InverseObject-Property (hasOffspring)). It follows that the hasOffspring edge in the rule graph is implied by the hasParent edge, so the hasOffspring edge has to be removed from the graph. Now that the rule graph is a skeleton, we have to verify it is admissible. With the implied edge removed, there are no cycles anymore and the property hierarchy fits the requirements. We can now build the rule's axiom and insert it into the KB. For that we have to traverse the graph and build the property chain that we will add as a subproperty of the head. Here is how to build a list of properties, w, for each nodes and edges:

- Node x:
  - No label in node.
  - Add hasParent to w.
- Node y:
  - No label in node.
  - Add hasSibling to w.
- Node z:
  - Add SubClassOf(ManObjectExistsSelf(instMan)) to the KB and instMan to w.
  - No successor.

The content of w is now hasParent, hasSibling, instMan. The last step is to insert the RIA modelling the rule in the ontology, SubObjectPropertyOf(SubObjectProp−ertyChain (hasParenthasSiblinginstMan)hasUncle).

6.1 Performance

FaCT++ 1.1.10 The rewriting technique we implemented being new, it had yet to be verified how the addition of the rules' axioms would influence the reasoner's performance, and how well it would scale. We conducted three sets of experiments to answer these questions: one in which the number of assertions grew, one where the number of rules and classes grew and in the last one the size of the whole ontology grows. The results are shown in Figure 3 where the displayed value is the average computation time of three independent runs to classify and realize the ontology(note that the Y axis has a logarithmic scale). However, the tendency is much more interesting than the values. The added axioms do not have a sizable effect on the computational properties of OWL and it has scaled well in our experiments. These results confirm the practical usability of the plug-in.

7 Conclusion and Future Work

In this paper, we presented a plug-in for the Protégé ontology editor that allows to edit rules, rewrite them as OWL axioms if they are admissible, or as SWRL rules otherwise, and insert them into an ontology. The plug-in includes

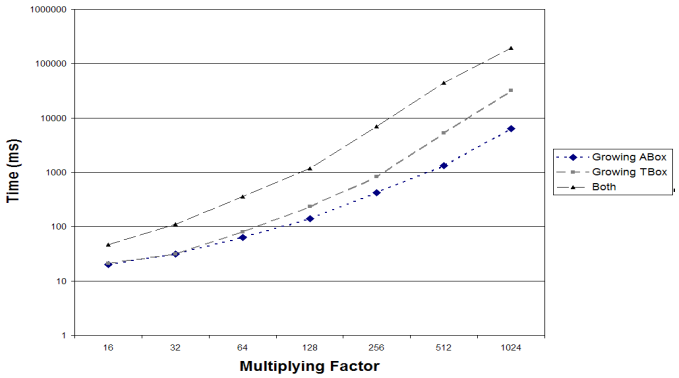

Fig. 3. Performance Graph.

interesting features such as a graphical visualization of the rules and a validation tool that shows the effect of a rule on the ontology. Rewriting the rules as OWL axioms let us avoid the restriction to DL-safety, that in turn allows one to get some nice inferences. Adding the non-rewritable rules as SWRL rules to the ontology allows the handling of arbitrary rules by the plug-in. It is to be noted that we also provide the functionalities of the plug-in in a command-line tool to allow the batch processing of ontologies. As future work, we plan to improve the plug-in so it can handle all datatypes restrictions valid in OWL 1.1 as atoms.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
2. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission (21 May 2004) Available at `http://www.w3.org/Submission/SWRL/`.
3. Motik, B., Sattler, U., Studer, R.: Query answering for owl-dl with rules. Journal of Web Semantics: Science, Services and Agents on the World Wide Web **3**(1) (JUL 2005) 41–60
4. Gasse, F., Sattler, U., Haarslev, V.: Rewriting rules into $\mathcal{SROIQ}$ axioms. Paper submitted (2008)
5. Various: Owl working group. W3C Working Group Available at `http://www.w3.org/2007/OWL/wiki/OWL_Working_Group`.
6. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006). Volume 4130 of Lecture Notes in Artificial Intelligence., Springer (2006) 292–297
7. Sirin, E., Parsia, B.: Pellet system description. In Parsia, B., Sattler, U., Toman, D., eds.: Description Logics. Volume 189 of CEUR Workshop Proceedings., CEUR-WS.org (2006)