A refinement operator for inducing Description Logics

Angelos Charalambidis and Stasinos Konstantopoulos

Institute of Informatics and Telecommunications,
NCSR 'Demokritos', Ag. Paraskevi 153 10, Athens, Greece
{acharal,konstant}@iit.demokritos.gr

Abstract. We present a new refinement operator that guides an ILP search through a lattice of a Horn clause representation of Description Logic (DL) clauses. Our operator dramatically narrows the search space by being aware of syntactic restrictions on the Horn representation stemming from both the syntax and semantics of DLs as well as from axioms present in the specific background of each experiment.

1 Introduction

Description logics (DL) are a family of logics that has found many applications in conceptual and semantic modelling, and is one of the key technologies behind *Semantic Web* applications. Despite, however, the rapid progress in inference methods for many-valued DL, there has been very limited success in applying machine learning methodologies to this family of logics, and especially to its more expressive members (such as those covering OWL and OWL 2) that are routinely used in web intelligence applications.

In this paper we first discuss previous work on applying ILP to learning DL (Section 2), and then present our formulation of the syntactic and semantic constraints that Horn clauses must comply to in order to constitute the Horn formulation of a valid DL clause (Section 3). Building on these results, we propose and experimentally validate a new ILP refinement operator more suitable for learning DL clauses (Section 4) and close the paper by drawing conclusions and outlining future research directions (Section 5).

2 Background

DL axioms are built using *descriptions* of classes (unary predicates), complex class expressions that can be recursively combined to build complex class descriptions. These descriptions might also involve relations (binary predicates), but relations themselves cannot be fully defined and described, as there is only limited support for relation descriptions and axioms.

Since ILP typically targets the domain of definite Horn clause programs, which is a different first-order fragment than that of DLs, applying ILP to learn DL descriptions is not straight-forward. In fact, many successful approaches

target *utilizing* DL background when constructing Horn clauses or other rules, rather than synthesising DL clauses [5, 9].

When it comes to synthesising DL clauses, the most obvious approach is to modify the ILP refinement operator so that a space of DL hypotheses is explored instead of a Horn space. Badea and Nienhuys-Cheng [2] propose such a top-down refinement operator. Although it addresses the over-specificity of the clauses learnt by previous bottom-up approaches [3], it is restricted to a simple DL without negation, numerical quantification, and relation composition. Lehmann & Hitzler Lehmann and Hitzler [8] and Ianone et al. Iannone et al. [4] address negation, but do not learn all descriptions possible in the more expressive DLs that cover the widely used OWL languages.

Coming from the opposite direction, one can also express DL constructs within Logic Programming, rather than adapting ILP systems to cover DL constructs that fall outside Logic Programming [6]. This approach uses Prolog meta-predicates to define DL constructs that fall outside the expressivity of pure LP. In other words, the full Prolog language (including meta-predicates) is used within the definitions of background predicates that implement DL semantics. In this manner the ILP algorithm has the building blocks needed to construct and evaluate DL statements *without* being given unrestricted access to meta-predicates for constructing clauses so that no fundamental assumption about the ILP setting is violated

3 Establishing Priors

At the core of ILP is a search through the hypotheses space, defined by the *refinement operator* and other pieces of *prior knowledge* such as the syntactic and semantic constraints that valid hypotheses must satisfy. Prior knowledge guides the search away from solutions that are known to be unsatisfiable or otherwise unwarranted, providing mechanisms for optimization and for enforcing conformance with a prior theoretical framework.

In our approach, there are two sources of prior knowledge: (a) DL syntax and semantics, so that all constructed hypotheses correspond to valid DL propositions, and (b) the axiomatization of the domain within which a theory is to be constructed, providing semantic restrictions that can optimize the search.

3.1 Constructing syntactically valid DL clauses

Description Logics are mainly characterized as the dyadic fragment of first-order logic (FOL), that is the fragment that only includes one and two-place predicates.[1] DL syntax has clauses connect a number of elementary constructs that all share a universally quantified variable. Some of these constructs may use a second (universally or existentially quantified) variable, but this variable is locally

[1] With some further restrictions that guarantee the efficiency of inference over the formalism, and that are not directly relevant to the current discussion.

scoped within the construct. Consider, for example, this statement equivalently formulated in DL, FOL, and our Horn notation [6]:

$$\exists \mathsf{hasLocomotive} \sqcap \forall \mathsf{hasCar}.\mathsf{Green} \sqsubseteq \mathsf{GreenTrain}$$

$$\forall x.\,(\exists y.\mathrm{hasLocomotive}(x,y) \wedge \forall y.\,(\mathrm{hasCar}(x,y) \rightarrow \mathrm{Green}(y)) \rightarrow \mathrm{GreenTrain}(x))$$

```
concept_select(X,'GreenTrain',Y) :-
   atleast_select(X,1,hasLocomotive,thing,Z),
   forall_select(Z,hasCar,'Green',Y).
```

As one can immediately see by comparing the three notations, the variable-free DL syntax provides a very natural way of restricting what can be written in it to the dyadic fragment of FOL, whereas extra checks are necessary in both FOL and our Horn syntax.

In the case of our Horn syntax, in particular, it must be checked that the first and last arguments of the body literals form a single in/out thread. Although threading can be achieved by mode declarations using the standard refinement operator, the requirement of a single thread can be encoded in mode declarations and needs to be implemented by pruning after invalid clauses have been constructed.

3.2 Consistency with background semantics

Besides learning hypotheses that can be cast into DL statements, it is also desired that the refinement operator constructs clauses that are consistent with the relations that hold between the domain-specific background predicates.

Background predicates in ILP are typically seen as the building blocks for constructing hypotheses, as they define the literals that are used in the bodies of the constructed clauses. Our approach partially departs from this tradition, as clauses are made up of the `*_select` predicates that define the framework or meta-theory within which theories are constructed. The first-order predicates of the theory itself (the concepts and relations of the DL-expressed domain) are *reified* into arguments for these predicates, and are treated as instances.

As such, the background theory is simultaneously represented as a first-order theory and as relations between the (reified) classes and relations of the theory. As an example, asserting that C is a sub-class of D asserts that all members of C are also members of D, but is also asserts a ground clause in the extension of the `concept_sub/2` predicate that keeps the class hierarchy:

```
concept_select(X,'Train',Y) :- concept_select(X,'GreenTrain',Y).
concept_sub('GreenTrain', 'Train').
```

Although not strictly necessary, `concept_sub/2` and related predicates affords our refinement operation direct access to the DL axioms that define the domain without having to parse the clauses of the `concept_select/3` predicate.

The refinement operator exploits this information to restrict the generated clauses to those that are consistent with these axioms. To make this more concrete, consider the existence of the classes Locomotive, Car, and Train in our domain and the knowledge that these are all disjoint as well as the knowledge that our target class, GreenTrain, is a sub-class of Train and the class Green is a sub-class of Car. Also consider that there are relations hasCar and hasLocomotive with domain Train and range Car and Thing respectively. In that case, the refinement operator will generate clauses with literals that respect the background constraints. For example, the clause

```
concept_select(A, 'GreenTrain', B) :-
    forall_select(A, 'hasCar', 'Green', B).
```

is a valid generated clause. On the other hand, the literal `forall_select(A, 'hasCar', Train, B)` must be avoided by the refinement operator because hasCar has a range that is disjointed to the class Train.

4 Implementation and Results

We have implemented a refinement operator that only generates clauses that observe the syntactic and semantic restrictions stated above.[2] The operator applies a top-to-bottom breadth-first search, ensuring that all body literals satisfy the single-thread constraint that produces clauses that map to variable-free DL syntax. The reified concept and relation names used in these literals are chosen after proving them against relation domain and range axioms, the concept hierarchy, and disjointness axioms.

Using the ALEPH [12] implementation of the PROGOL algorithm [11] and our implementation of DL inference described above, we tested our approach on a variation [6] of the famous trainspotting machine-learning problem [7] and on the Iris dataset [1]. On the Iris dataset the experiment was organized as three different learning tasks, one for each of the three targets of the classification task. The induction run statistics are collected in Table 1.

It is straightforward to observe from the results that in all experiments the custom refinement operator constructs a dramatically smaller number of clauses: the standard operator generates a large number of clauses that either do not comply with the syntactic restrictions (Section 3.1) and are pruned after construction or do not comply with the semantic restrictions of the domain (Section 3.2) and are known in advance to not cover any positive examples.

In terms of execution time, the total execution time is also reduced in the same amount as the reduction of the visited nodes. We have derived the average time of a single reduction from the total visited nodes and the total execution

[2] Please cf. `http://sourceforge.net/projects/yadlr` The instance-based engine described here is `pl/prodlr.pl`. The refinement operator described is in `pl/dllearn.pl` along with the standard refinement setup and pruning. The background and datafiles for the experiments described here are under `examples/`

Table 1. The number of nodes visited before constructing identical theories using the refinement operator described here and our previous pruning-based implementation [6]. In pruning, we show the number constructed clauses and the number of those that has been pruned.

	Nodes visited			Avg. Time per reduction (ms)	
Priors enforced	Pruning	Refinement	Gain	Pruning	Refinement
Trains	41/23	11	73.17%	4.19	1.45
Iris Setosa	200/145	9	95.50%	2.10	4.00
Iris Versicolour	24698/20180	742	96.99%	1.28	2.81
Iris Virginica	11202/9118	351	96.86%	1.32	2.62

time. The reduction time includes the construction of a new refinement, the evaluation of the clause and the application of the clause to the positive and negative examples. Notice that in the results presented in Table 1 we have also a better performance in average times on the trains dataset. On the other hand, the average times when we are using the custom operator are higher on the Iris datasets. The reason for this is that in the Iris dataset the priors only involve syntactic constraints and there are no prior semantic grounds for rejecting a syntactically valid clause. For this reason, the more complex calculations in the custom refinement clause make each refinement step more expensive, although the overall induction time is greatly improved using the custom refinement operator due to the smaller number of constructed and evaluated nodes.

The learning task and experimental setup is as described previously by Konstantopoulos and Charalambidis [6], the only difference being the application of the refinement operator introduced here. In that work, conformance to DL syntax and semantics was only partially achieved via mode declarations and had to be complemented pruning, whereas here we are able to altogether avoid refining into ill-formed clauses.

5 Conclusions

In this paper we pursue a line of research on expressing DL class descriptions *within the Logic programming framework,* allowing for the direct application of well-tried and highly-optimized ILP systems to the task of synthesising class descriptions. More specifically, we investigate a principled way of extracting more ILP bias from DL models in order to increase the efficiency of ILP runs.

This investigation led to the definition and implementation of a new refinement operator that is empirically shown to dramatically narrow the search space by being aware of restrictions on the Horn representation stemming from both the syntax and semantics of DLs as well as from axioms present in the specific background of each experiment.

Future plans include the extension of the DL-expressed domain axioms that contribute to ILP prior bias, in order to further restrict the search space as well as the investigation of appropriate evaluation functions, in the vein of recent research by Lisi and Straccia [10].

References

[1] Asuncion, A., Newman, D.J.: UCI machine learning repository. University of California, Irvine, School of Information and Computer Sciences (2007)

[2] Badea, L., Nienhuys-Cheng, S.H.: A refinement operator for description logics. In: Cussens, J., Frisch, A.M. (eds.) Proceedings of ILP 2000, LNCS 1866, pp. 40–59. Springer Verlag, Berlin/Heidelberg (2000)

[3] Cohen, W.W., Hirsh, H.: Learning the CLASSIC description logic: Theoretical and experimental results. In: Proc 4th Intl Conf on Principles of Knowledge Representation and Reasoning. pp. 121–133 (1994)

[4] Iannone, L., Palmisano, I., Fanizzi, N.: An algorithm based on counterfactuals for concept learning in the Semantic Web. Journal of Applied Intelligence 26(2), 139–159 (Apr 2007)

[5] Iglesias, J., Lehmann, J.: Towards integrating fuzzy logic capabilities into an ontology-based inductive logic programming framework. In: Proc. of the 11th International Conference on Intelligent Systems Design and Applications ISDA, 2011 (2011)

[6] Konstantopoulos, S., Charalambidis, A.: Formulating description logic learning as an inductive logic programming task. In: Proceedings of 2010 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2010), July 18–23, Barcelona. IEEE (Jul 2010)

[7] Larson, J., Michalski, R.S.: Inductive inference of VL decision rules. ACM SIGART Bulletin 63, 38–44 (Jun 1977)

[8] Lehmann, J., Hitzler, P.: A refinement operator based learning algorithm for the $\mathcal{ALC}$ description logic. In: Proceedings of ILP 2007, LNCS 4894, pp. 147–160. Springer Verlag, Berlin/Heidelberg (Feb 2008)

[9] Lisi, F.A., Malerba, D.: Bridging the gap between horn clausal logic and description logics in inductive learning. In: Proc. 8th Congress of the Italian Association for Artificial Intelligence, Pisa, Italy, 23-26 Sep, 2003. LNCS 2829, pp. 53–64. Springer-Verlag, Berlin/Heidelberg (2003)

[10] Lisi, F.A., Straccia, U.: Can ilp deal with incomplete and vague structured knowledge? In: Proceedings of ILP 2011, short papers (2011)

[11] Muggleton, S.: Inverse entailment and Progol. New Generation Computing 13, 245–286 (1995)

[12] Srinivasan, A.: The Aleph Manual. `http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/` (Last update: 13 Mar 2007)