Position Paper: Ontological Logic Programming *

Murat Şensoy, Geeth de Mel, Wamberto W. Vasconcelos, and Timothy J. Norman

Department of Computing Science, University of Aberdeen, AB24 3UE, Aberdeen, UK
{m.sensoy,g.demel,w.w.vasconcelos,t.j.norman}@abdn.ac.uk

Abstract. In this paper, we propose a novel approach that combines logic programming with ontological reasoning. The proposed approach enables the use of ontological terms directly within logic programs. We demonstrate the usefulness of the proposed approach using a case-study of sensor-task matchmaking.

1 Introduction

Description Logic (DL) is a decidable fragment of First Order Logic (FOL) [2]. It constitutes the formal background for OWL-DL, the decidable fragment of the Web Ontology Language (OWL) [7]. However, DL is not sufficient on its own to solve many real-life problems. For example, some rules may not be expressed in DL. In order to represent rules in an ontology, rule languages such as Semantic Web Rule Language (SWRL) [1] have been proposed. In the design of Semantic Web languages, decidability has been one of the main concerns. To achieve decidability, these languages enforce limitations on expressiveness. OWL ensures decidability by defining its DL equivalent subset; similarly we can ensure decidability of SWRL using only DL-safe rules [4]. Existing reasoners such as Pellet [6] provide ontological reasoning services based on these restrictions. However, because of these limitations, many logical axioms and rules cannot be expressed using OWL-DL and SWRL [1].

On the other hand, languages like Prolog [8] provide very expressive declarative Logic Programming (LP) frameworks. Unlike OWL and SWRL, Prolog adopts the closed-world assumption through *negation as failure* and enables complex data structures and arbitrary programing constructs [8]. In this paper, we propose Ontological Logic Programming (OLP)[1] , a novel approach that combines LP with DL-based ontological reasoning. An OLP program can dynamically import various ontologies and use the terms (i.e., classes, properties, and individuals) in these ontologies directly within an OLP program. The interpretation of these terms are delegated to an ontology reasoner during interpretation of the OLP program.

* This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

[1] OLP's source code is publicly available at http://olp-api.sourceforge.net

2 Ontological Logic Programming

Figure 1 shows the stack of technologies and components used to interpret OLP programs. At the top of the stack, we have the OLP interpreter, which sits on top of a LP layer. The LP layer is handled by a Prolog engine. The Prolog engine uses two different knowledge bases; one is a standard Prolog knowledge base of facts and clauses while the other is a semantic knowledge base composed of OWL-DL ontologies and SWRL rules. Pellet [6] has been used as a DL reasoner to interface between the Prolog engine and the semantic knowledge base.

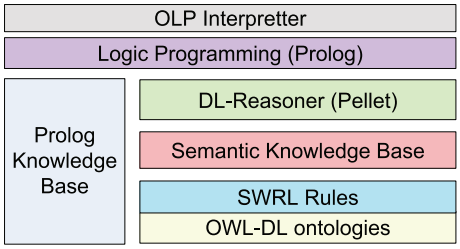

Fig. 1. OLP Stack.

Our choice of LP language is Prolog and in this work, we use a pure Java implementation, *tuProlog* [5]. The OLP interpreter is a Prolog meta-interpreter with a set of OLP-specific predicates. Figure 2 shows a simplified version of the OLP interpreter used to evaluate OLP programs through the $eval/1$ predicate. While interpreting OLP programs, the system behaves as if it is evaluating a standard Prolog program until it encounters an ontological predicate. In order to differentiate ontological and conventional predicates, we use name-space prefixes separated from the predicate name by a colon, i.e., ":". For example, if W3C's wine ontology[2] is imported, we can directly use the ontological predicate *vin:hasFlavor* in an OLP program without the need to define its semantics, where *vin* is a name-space prefix that refers to *http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#*. This name-space prefix is defined and used in the wine ontology. The Prolog knowledge base does not have any knowledge about ontological predicates, since these predicates are not defined in Prolog, but described separately in an ontology, using DL [2]. In order to interpret ontological predicates, the OLP interpreter needs ontological reasoning services provided by a DL reasoner. Hence, we have a DL reasoning layer below the LP layer. The interpreter accesses the DL reasoner through the $dl\_reasoner/1$ predicate as shown in Figure 2. This predicate is a reference to a Java method, which queries the reasoner and evaluates the ontological predicates based on ontological reasoning. OLP uses two disjoint knowledge bases. A Prolog knowledge base is used to store, modify and reason about non-ontological facts and clauses (e.g., rules), while a semantic knowledge base is used to store, modify and reason about ontological predicates and semantic rules. The semantic knowledge base is based on a set of OWL-DL ontologies, dynamically imported by OLP using *import* statements. Some rules are associated with these ontologies using SWRL [1]. Above

[2] It is located at `http://www.w3.org/TR/owl-guide/wine.rdf` and imports W3C's food ontology located at `http://www.w3.org/TR/owl-guide/food.rdf`.

the ontologies and the semantic rules, we have Pellet [6] as our choice of DL reasoner. It is used to infer facts and relationships from the ontologies and semantic rules transparently.

```
:- op(550,xfy,':').
eval((O:G)):- dl_reasoner((O:G)).
eval(assert((O:G))):- assert_into_ontology((O:G)).
eval(retract((O:G))):- retract_from_ontology((O:G)).
eval(not(G)):- not(eval(G)).
eval((G1,G2)):- eval(G1),eval(G2).
eval((G1;G2)):- eval(G1);eval(G2).
eval(G):- not(complex(G)),(clause(G,B),eval(B);
          not(clause(G,_)),call(G)).
complex(G):- G=not(_);G=(_,_);G=(_;_);G=(_:_);
             G=assert(_:_);G=retract(_:_).
```

Fig. 2. Prolog meta-interpretter for OLP interpreter.

During the interpretation of an OLP program, when a predicate in *prefix:name* format is encountered, the DL reasoner below the LP layer in the OLP stack is queried to get direct or inferred facts about the predicate in the underlying ontologies. For example, when the meta-interpreter encounters *vin:hasFlavor(D,R)* during its interpretation of an OLP program, it queries the DL reasoner, because *vin:hasFlavor* is an ontological predicate. The *hasFlavor* predicate is defined in the wine ontology, so the reasoner interprets its semantics to infer direct and derived facts about it. Using this inferred knowledge, the variables D and R are unified with the appropriate terms from the ontology. Then, using these unifications, the interpretation of the OLP program is resumed. Therefore, we can directly use the concepts and properties from ontologies while writing logic programs and the direct and derived facts are imported from the ontology through a reasoner when necessary. In this way, OLP enables us to combine the advantages of logic programming (e.g., complex data types/structures, negation by failure and so on) and ontological reasoning. Moreover, logic programming aspect enables us to easily extend the OLP interpreter so as to provide, together with answers, explanations of the reasoning which took place.

3 Case-Study

In order to ground the description of OLP, in this section we introduce a real-world problem domain and shows how OLP has been used to provide an effective solution to it. The International Technology Alliance[3] (ITA) is a research program initiated by the UK Ministry of Defence and the US Army Research Laboratory. ITA focuses on the research problems related to wireless and sensor networks. One of these research problems is the selection of appropriate sensing resources for *Intelligence, Surveillance, Target Acquisition and Reconnaissance (ISTAR)* tasks[4]. In order to solve this problem, we have previously implemented a system called *Sensor Assignment to Missions (SAM)* [3]. Here, we demonstrate how SAM has been significantly improved using OLP.

[3] http://en.wikipedia.org/wiki/International_Technology_Alliance
[4] http://en.wikipedia.org/wiki/ISTAR

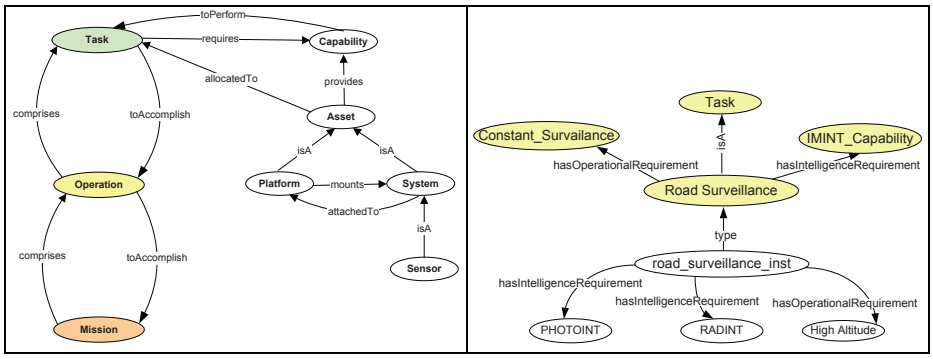

Fig. 3. The ISTAR ontology on the left and a task instance example on the right.

3.1 ISTAR Tasks and Sensing Resources

We show, in Figure 3, a part of the ontology for the ISTAR domain. In the ontology, the *Asset* concept represents the resources that could be allocated to tasks. The *Platform* and *System* concepts are both assets, but systems may be attached to platforms. Sensors are a specialisation of systems. A sensor needs to be mounted on a platform to work properly. On the other hand, not all platforms can mount every type of sensors. For example, to be used, a radar sensor must be mounted on Unmanned Aerial Vehicles (UAVs), however only specific UAVs such *Global Hawk* can mount this type of sensors.

A task may require capabilities, which are provided by the assets. In order to achieve a task, we need to deploy specific assets that provide the required capabilities. Capability requirements of a task are divided into two categories: the first concerns operational capabilities provided by the platforms, and the second concerns intelligence capabilities provided by the sensors attached to a platform. Figure 3 shows *Road Surveillance* task, which has one operational requirement, namely *Constant Surveillance*, and one intelligence requirement, namely *Imagery Intelligence* (IMINT). As shown in the figure, an instance of this task is then defined with two more intelligence requirements (*Radar Intelligence* and *Photographical Intelligence*) and an additional operational requirement (*High Altitude*). We use the term *Deployable Configuration* to refer a set of assets required to achieve a task. A deployable configuration of a task is composed of a deployable platform and a set of sensors. A deployable platform provides all operational capabilities required by the task. Similarly, the sensors in the deployable configuration provide all the intelligence capabilities required by the task. Furthermore, the deployable platform should have an ability to mount these sensors. Therefore, there is a dependency between the platform and the sensors in a deployable configuration.

3.2 Resource-Task Matchmaking using OLP

The first version of SAM [3] uses a minimal set covering algorithm to compute deployable configurations for an ISTAR task. That algorithm enumerates all possible sets of asset types so that each set has at most n members. Then, a set is regarded as a deployable configuration of the task if it satisfies all the requirements. Here, we extend SAM via an OLP program shown in Figure 4 to compute deployable configurations.

The OLP program is a Prolog program, where concepts and properties from the underlying ontologies are referenced directly. The *getConfigurations* predicate computes deployable configurations for a specific task. Each sensor must be carried by a deployable platform that provides all of the operational requirements of the task (e.g., constant surveillance). If a sensor cannot be carried by a deployable platform, there is no point in considering deployable configurations with that sensor type. Using this knowledge, a tailored and efficient matchmaker can be employed. This matchmaker first identifies the deployable platforms that meet the requirements of the task. Once many possibilities are narrowed down by determining deployable platforms, the sensor types that provide the intelligence capabilities required by the task are determined incrementally so that those sensors can be mounted on the deployable platforms.

```
%import http://www.csd.abdn.ac.uk/~murat/ita/istar.owl
getConfigurations(T,[P|S]):-
   deployablePlatform(T,P),
   extendSolution(T,P,[],S).
deployablePlatform(T,P):-
   istar:'Platform'(P),
   not((istar:'requireOperationalCapability'(T,C),
        not(istar:'provideCapability'(P,C)))).
extendSolution(T,P,Prev,Next):-
   requireSensor(T,P,Prev,X),
   istar:'mounts'(P,X),
   A=[X|Prev],
   extendSolution(T,P,A,Next).
extendSolution(T,P,S,S):-
   not(requireCapability(T,P,S,_)).
requireSensor(T,P,S,X):-
   requireCapability(T,P,S,C),
   istar:'provideCapability'(X,C).
requireCapability(T,P,S,C):-
   istar:'requireCapability'(T,C),
   not(provideCapability(S,C)),
   not(provideCapability([P],C)).
provideCapability([Y|Tail],C):-
   istar:'provideCapability'(Y,C),!;
   provideCapability(Tail,C).
```

Fig. 4. OLP program to compute deployable configurations.

We have compared the OLP-based matchmaker with the exhaustive search approach in terms of time consumption. For this purpose, we randomly created 908 tasks using the ISTAR ontology. Figure 5 shows our results, where the *x-axis* is the maximum number of items in deployable configurations and *y-axis* is the average time consumed by each approach to find all of the deployable configurations of a task. When the maximum size of deployable configurations increases, the OLP-based approach outperforms the exhaustive search approach significantly; time consumption of the exhaustive search increases exponentially while that of the proposed approach looks mostly linear. These results are intuitive because the OLP program of Figure 4 is based on the idea that the search space can be significantly reduced using domain knowledge (i.e, dependencies between sensors and platforms; not every types of sensors can be used with a specific type platform). Using this principle, at each iteration, it rules out many combinations and significantly reduces the time required to compute deployable configurations.

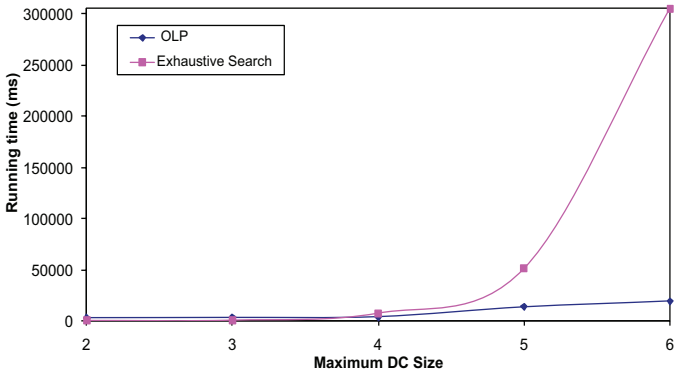

Fig. 5. Comparison between OLP program of Figure 4 and the exhaustive search algorithm to find deployable configurations.

4 Conclusions

In this paper, we have proposed a novel tool that combines Logic Programming with Ontological Reasoning. Unlike similar approaches in the literature, our approach delegates interpretation of ontological predicates to an ontology reasoner during the execution of logic programs. Hence, it takes the full advantage of both ontological reasoning and logic programming without any compromise in expressiveness. Using a case-study, we have demonstrated how the proposed approach can be used to solve sensor-task matchmaking problem in an efficient and practical way.

References

1. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004. http://www.w3.org/Submission/SWRL.
2. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
3. M. Gomez, A. Preece, M. P. Johnson, G. Mel, W. Vasconcelos, C. Gibson, A. Bar-Noy, K. Borowiecki, T. Porta, D. Pizzocaro, H. Rowaihy, G. Pearson, and T. Pham. An ontology-centric approach to sensor-mission assignment. In *Proceedings of the 16th international conference on Knowledge Engineering (EKAW'08)*, pages 347–363, 2008.
4. P. Haase and B. Motik. A mapping system for the integration of owl-dl ontologies. In *IHIS '05: Proceedings of the first international workshop on Interoperability of heterogeneous information systems*, pages 9–16, New York, NY, USA, 2005. ACM.
5. G. Piancastelli, A. Benini, A. Omicini, and A. Ricci. The architecture and design of a malleable object-oriented Prolog engine. In R. L. Wainwright, H. M. Haddad, R. Menezes, and M. Viroli, editors, *23rd ACM Symposium on Applied Computing (SAC 2008)*, pages 191–197, 2008.
6. E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Web Semant.*, 5(2):51–53, 2007.
7. M. K. Smith, C. Welty, and D. L. McGuinness. OWL: Web ontology language guide, February 2004.
8. L. Sterling and E. Shapiro. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1986.