

Ontological Logic Programming *

Murat Şensoy
Computing Science
University of Aberdeen
AB24 3UE, Aberdeen, UK
m.sensoy@abdn.ac.uk

Geeth de Mel
Computing Science
University of Aberdeen
AB24 3UE, Aberdeen, UK
g.demel@abdn.ac.uk

Wamberto W.
Vasconcelos
Computing Science
University of Aberdeen
AB24 3UE, Aberdeen, UK
w.w.vasconcelos@abdn.ac.uk

Timothy J. Norman
Computing Science
University of Aberdeen
AB24 3UE, Aberdeen, UK
t.j.norman@abdn.ac.uk

ABSTRACT

In this paper, we propose Ontological Logic Programming (OLP), a novel approach that combines logic programming with ontological reasoning. The proposed approach enables the use of ontological terms (i.e., individuals, classes and properties) directly within logic programs. The interpretation of these terms are delegated to an ontology reasoner during the interpretation of the program. Unlike similar approaches, OLP makes use of the full capacity of both the ontological reasoning and logic programming. We evaluate the computational properties of OLP in different settings and show that its performance can be significantly improved using caching mechanisms. Furthermore, using a case-study, we demonstrate the usefulness of OLP in real-life settings.

Categories and Subject Descriptors

I.2.4 [Artificial Intelligence]: Knowledge Representation Formalisms and Methods – semantic networks, ontologies

General Terms

Languages, Performance

Keywords

Semantic Web, Logic Programming

*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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WIMS '11, May 25-27, 2011 Sogndal, Norway

Copyright 2011 ACM 978-1-4503-0148-0/11/05 ...\$10.00.

1. INTRODUCTION

Description Logic (DL) is a decidable fragment of First Order Logic (FOL) [4] and constitutes the background for OWL-DL, the decidable fragment of the Web Ontology Language (OWL) [22]. 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 [11]. Existing reasoners such as Pellet [21] 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 [23] provide very expressive declarative Logic Programming (LP) frameworks. Unlike OWL and SWRL, Prolog adopts the closed-world assumption¹ through *negation as failure* and enables complex data structures and arbitrary programming constructs [23]. In this paper, we propose Ontological Logic Programming (OLP)², 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. By enhancing logic programming with ontological reasoning, OLP offers the following advantages:

1. **Expressiveness:** OLP combines the expressiveness of DL and LP. Hence, the limitations of OWL-DL are compensated by the high expressiveness of LP.
2. **Convenience:** Many researchers and developers are more familiar with LP languages than with DL formalisms. OLP enables DL reasoning to be used transparently within a logic program.
3. **Reuse of Domain Knowledge:** In logic programs, domain knowledge is encoded within the program, often in an ad-hoc

¹http://en.wikipedia.org/wiki/Closed_world_assumption

²OLP's source code is publicly available at <http://olp-api.sourceforge.net>

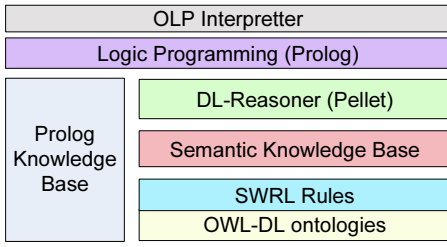


Figure 1: OLP Stack.

manner. OLP enables domain knowledge to be defined in a set of ontologies in a standard way. These ontologies may then be easily used by different OLP programs.

4. **Conciseness:** OLP programs are far more concise than equivalent standard logic programs. This is because OLP programs use domain ontologies to reason about domain knowledge, while standard logic programs require this domain knowledge to be encoded within the program. Consider the rule *a person can drive a vehicle only if he/she has a driving license*. In a logic program, in order to express this rule, semantics and facts about the terms *person*, *transportation vehicle*, *driving*, and *driving license* have to be formalised within the logic program. This means that the program has to be much longer than the rule to be expressed. On the other hand, an OLP program simply imports appropriate ontologies that contain domain knowledge about these terms, and expresses the rule concisely using these terms.
5. **Reuse of Logic Programs:** Logic programming, notably Prolog, has been used for decades to develop many AI applications such as expert systems, planning systems, theorem provers, and so on. The proposed combination would allow “legacy” AI systems to take advantage of more recent Semantic Web developments, namely, open standards for knowledge representation with publicly available ontologies, as well as efficient reasoning mechanisms, without the need for re-implementation.

In this paper, we present Ontological Logic Programming in Section 2 both in terms of its architecture, and how OLP interacts with the underlying semantic knowledge. In Section 3, we analyse the performance of OLP with respect to different strategies for incorporating ontological reasoning into logic programming. We then present a detailed real-life case-study from the sensor resource management domain in Section 4. In Section 5, we discuss the contributions of our approach compared with key related research. Finally, we present our conclusions in Section 6.

2. ONTOLOGICAL LOGIC PROGRAMMING

We present OLP in two stages. First, we introduce the OLP stack and describe how OLP interprets logic programs using semantic knowledge. We then discuss in detail how OLP modifies the underlying semantic knowledge and accesses semantic reasoning services.

2.1 Architecture

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 [21] has been used as a DL reasoner to interface between the Prolog engine and the semantic knowledge base.

Our choice of LP language is Prolog and in this work, we use a pure Java implementation, *tuProlog* [18]. The OLP interpreter is a Prolog meta-interpreter with a set of OLP-specific predicates, described in Section 2.2. 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 ontology³ 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 [4]. 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 the ontologies and the semantic rules, we have Pellet [21] as our choice of DL reasoner. It is used to infer facts and relationships from the ontologies and semantic rules transparently.

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,

³Located 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>.

```

:- 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(_:_).

```

Figure 2: OLP interpreter in Prolog.

together with answers, explanations of the reasoning which took place.

Namespaces play an important role in OLP. That is why, the OLP Application Programming Interface (API) and Integrated Development Environment (IDE) provides various utilities to define new namespaces, list namespaces already defined in the imported ontologies, and search ontological predicates within these namespaces using regular expressions. Furthermore, OLP IDE provides IntelliSense⁴ capabilities to enhance programming experience by auto-completion and lookahead features. Figure 3 shows a snapshot of OLP IDE, with a simple program. The program starts with an import statement for the wine ontology. Available namespaces can be listed in a new window using *NS* button on the menu. This window is shown just below the menu and can also be used to search for ontological terms within these namespaces using regular expressions. In the middle of the figure, an IntelliSense pop-up window is shown next to *food:c*; it lists all ontological terms starting with *c* in the name-space *food*.

Lastly, it is important to explain the effects of Prolog’s backtracking mechanism on our interpreter of Figure 2. The meta-interpreter undergoes backtracking in the standard fashion [3], exhaustively attempting to find a solution to a query *eval(G)*, trying different clauses in turn – the clauses, with the exception of *complex(G)*, are mutually exclusive, due to the patterns they have in their head goals. Prolog also tries different ways to prove the goals in the body of a clause, backtracking when one of them fails, and attempting to prove the previous goal again (hopefully obtaining a different set of values for its variables), until a solution is found to the last goal of the clause’s body. We control the effects of backtracking on the invocation of the external DL reasoner, namely, the predicate *dl_reasoner(O:G)* in the first clause. We rely on the termination properties of our reasoner, Pellet, and the limited expressiveness of DL (for instance, circular definitions cannot be expressed), to compute *all* possible solutions for *O:G* upon the first invocation of the predicate, and to produce these solutions one at a time upon backtracking.

⁴<http://en.wikipedia.org/wiki/IntelliSense>

2.2 Semantic Knowledge and OLP

OLP not only uses the semantic knowledge within ontologies, but it may also modify this knowledge by importing new ontologies, and adding or removing concepts, roles, individuals and facts (i.e., RDF statements [22]). For this purpose, we provide OLP-specific predicates, some of which are listed in Table 2. Here, we outline how OLP may be used to modify the semantic knowledge base:

- **Importing ontologies.** In a classical Prolog program, domain knowledge is encoded as a part of the Prolog knowledge base. To facilitate the reuse of standardised domain ontologies, OLP enables Prolog programs to directly use predicates defined in ontologies. An OLP program may import a number of ontologies to access the domain knowledge encoded within them. We provide two mechanisms to do this. First, at the beginning of an OLP program, lines starting with *%import* are interpreted as an instruction to import an ontology located at a specific URI (note that these lines start with *%*, so they are regarded as comments by the Prolog engine). Second, the *import_ontology* predicate can be directly used within an OLP program to dynamically import new ontologies.
- **Addition and removal of statements.** As shown in Figure 2, the OLP interpreter evaluates *assert* and *retract* predicates differently depending on whether these are ontological and non-ontological facts. If *assert* is used with an ontological statement as in *assert(vin:‘Wine’(olp:x))*, then the *assert_into_ontology* predicate is used by the interpreter to add this statement to the semantic knowledge base. That is, the semantic knowledge base is modified by declaring *olp:x* as an instance of the *Wine* concept. On the other hand, if *assert* is used with non-ontological predicates as in *assert(served(vin:‘TaylorPort’))*, a new fact is added to the underlying Prolog knowledge base. It should be noted that the addition of a new statement to the semantic knowledge base may make it inconsistent. For example, addition of the statement *rdf:subConceptOf(vin:‘Wine’, food:‘Fruit’)* results in an inconsistent semantic knowledge base, because *Wine* and *Fruit* concepts in the wine ontology are defined as disjoint concepts. Therefore, before adding the statement, *assert_into_ontology* checks whether the addition would result in an inconsistency. If the addition would result in an inconsistency, *assert_into_ontology* returns *false* without adding the statement. Otherwise, it modifies the knowledge base and returns *true*. The *retract* predicate works in a similar way: ontological facts are removed from the underlying semantic knowledge base using the *retract_from_ontology* predicate, while others are removed directly from the Prolog knowledge base.
- **Addition and removal of individuals.** New individuals can be created using the *create_individual* predicate. For example, *create_individual(vin:‘SoftWine’)* creates the individual *SoftWine* within the name-space *vin* as an instance of *owl:Thing*. Then using *assert(vin:‘Wine’(vin:‘SoftWine’))*, we can declare that *vin:‘SoftWine’* is a wine. On the other hand, using the *remove_individual* predicate, we can remove an individual and all statements about that individual from the semantic knowledge base (e.g., *remove_individual(vin:‘SoftWine’)*).
- **Addition and removal of concepts.** Through the *create_concept* predicate, a new OWL-DL concept can be created based on

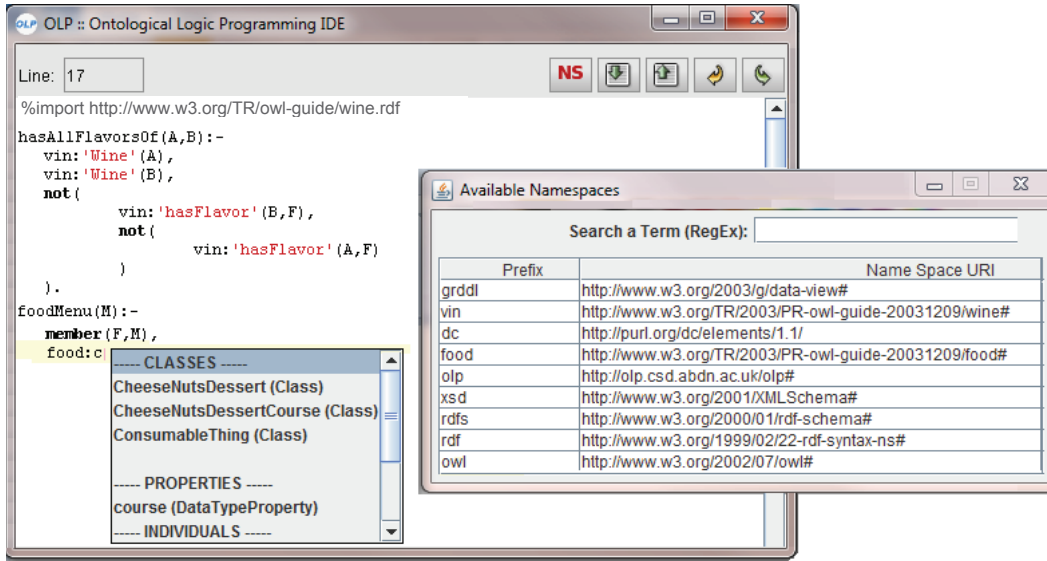


Figure 3: A snapshot of OLP Integrated Development Environment.

Table 1: Simple concept description examples.

Concept Description	Satisfiable
vin:'Wine'	yes
(vin:'Wine'; food:'Fruit')	yes
(vin:'Wine', food:'Fruit')	no
enum(vin:'TaylorPort', food:'ThompsonSeedless')	yes
(vin:'Wine', not(enum(vin:'TaylorPort')))	yes
value(vin:'hasFlavor', vin:'Delicate')	yes
all(inverse(vin:'hasFlavor'), vin:'SauvignonBlanc')	yes
some(vin:'hasMaker', vin:'SaucelitoCanyon')	yes

a DL class description. If the described concept is not satisfiable, the predicate returns *false* without creating the concept; otherwise it returns *true* after creating the concept. A concept description is an OWL-DL class expression [22], which can be a single concept name, a restriction on properties, or created using the intersection or the union of two class expressions or the complement of a class expression. A restriction on a property can be specified through *someValuesFrom*, *allValuesFrom*, *minCardinality*, *maxCardinality* and exact *cardinality* restrictions [22]. The inverse of a property can also be used in a concept description. Moreover, a concept can be described by enumerating all of its instances; such classes are called enumerated classes [22]. Table 1 shows examples of concept descriptions. OLP also allows the removal of concepts from the semantic knowledge base using *remove_concept* predicate. When this predicate is used, not only the concept but also all statements about the concept are removed from the semantic knowledge base.

In addition to the reasoning capabilities provided by Prolog, OLP provides access to the ontological reasoning services provided by the underlying DL reasoner. These are summarised as follows:

- **Ontology consistency checking.** OLP enables consistency checking of the underlying semantic knowledge base through *has_consistent_ontology* predicate. This predicate returns *true* if the underlying semantic knowledge base is consistent; otherwise it returns *false*.

- **Concept satisfiability checking.** In order to test the satisfiability of a concept description, *is_satisfiable_concept* predicate should be used; it returns *true* if the described concept is satisfiable; otherwise it returns *false*.
- **Concept subsumption checking.** In order to test if a named concept c_i subsumes another c_j ($c_j \sqsubseteq c_i$), the ontological predicate *rdfs:subClassOf*(c_j, c_i) should be used. For example, when *rdfs:subClassOf*(*vin:'Riesling', A*) is found, the OLP interpreter unifies *A* with the super-concepts of *Riesling*, such as *food:'PotableLiquid'*, *vin:'WhiteWine'*, *vin:'Wine'*, and so on.
- **Class equivalence checking.** In order to check equivalence of two named concepts c_i and c_j , the ontological predicate *rdfs:equivalentClass*(c_i, c_j) should be used. For example, when the predicate *rdfs:equivalentClass*(*vin:'Riesling', A*) is found, *A* is unified with the concepts equivalent to *Riesling* if any exist.
- **Class instance checking.** OLP provides two different ways to reason about the instances of a specific concept. The first way is to use the class name as an ontological predicate; for example, when *vin:'Wine'(I)* is found, the OLP interpreter unifies *I* with instances of the *Wine* concept in the *vin* namespace. The second way is to use the ontological predicate *rdf:type*; for example, when *rdf:type(I, vin:'Wine')* is used, the OLP interpreter also unifies *I* with instances of *Wine*.

3. PERFORMANCE OF OLP

In this section, we evaluate the computational properties of the OLP by executing a set of queries against W3C's wine ontology. Queries are created randomly and with increasing complexity (i.e., number of ontology predicates are increased continuously up to a particular threshold) and negation to represent failure (e.g., any wine which is not sweet). All the experiments are run on a PC with 2.16 GHz Intel Core Duo processor and 2GB of RAM.

As described in Section 2, OLP works by communicating with a DL reasoner whenever it encounters an ontological term in order

Table 2: OLP-specific predicates and their intended meaning.

Predicate	Description
import_ontology(URI).	Imports an ontology located at a given URI.
add_namespace(Prefix, URI).	Creates a name-space with a given prefix and URI.
create_individual(Name).	Creates an instances of <i>owl:Thing</i> class in the semantic knowledge base with a given name.
remove_individual(Name).	Removes an individual and all statements about that individual from the semantic knowledge base.
assert_into_ontology(Stmt).	Given an RDF statement, it returns <i>true</i> after adding the statement into the semantic knowledge base if this addition results in a consistent ontology; otherwise it returns <i>false</i> without adding the statement.
retract_from_ontology(Stmt).	Removes an RDF statement from the underlying semantic knowledge base.
is_satisfiable_concept(Desc).	Given a concept description, returns <i>true</i> if the concept is satisfiable; otherwise returns <i>false</i> .
has_consistent_ontology.	Returns <i>true</i> if the underlying semantic knowledge base is consistent; otherwise returns <i>false</i> .
create_concept(Name, {eq sub}, Desc).	Given a concept name and description, creates a new OWL-DL concept equivalent to or subsumed by the described concept in the semantic knowledge base.
remove_concept(Name).	Removes an OWL-DL concept and all statements related to this concept from the underlying semantic knowledge base.
save_ontology(Filename).	Saves the existing semantic knowledge base as an OWL ontology with a given name.

to resolve it. We refer to this mode as *online*. Purely for evaluation purposes, we run OLP in another mode. In this second mode, using the reasoner, we load all of the direct and inferred facts from the ontology into the Prolog’s standard knowledge base and then execute the queries with respect to Prolog’s knowledge base instead of interacting with the reasoner during the execution; we refer to this mode as *offline* hereafter. It is important to note that, when offline mode is used, the Prolog knowledge base is populated with all of the direct and inferred facts from the ontology before executing any query. We executed four random query sets against the wine ontology. In order to analyse offline and online modes for different ontology sizes, we extended the size of the wine ontology by introducing new properties and concepts. We analysed the online and offline modes in terms of *load time*, *reasoner access time*, and *average query time*. The load time is the time spend to load ontological facts into the Prolog knowledge base in offline mode. On the other hand, the reasoner access time is the time spent by the reasoner during the execution of queries in online mode. Lastly, *query time* for a query is the time spent by the Prolog engine within the OLP stack to answer the query and excludes *load* and *reasoner access time*. That is, the total time required to execute a query and retrieve all of its answers is *load_time + query_time* for offline mode, while it is *reasoner_access_time + query_time* for online mode.

The graph in Figure 4 shows the average query time in each mode as the ontology size increases. Clearly, the online mode significantly outperforms the offline mode. Furthermore, the performance difference increases dramatically while the ontology size increases. This is mainly due to the fact that offline mode introduces redundancies by loading the whole ontological knowledge and as the size of its knowledge base increases, the performance of the Prolog engine decreases. We also analysed the effect of query complexity on query time for online and offline modes, and the results of this

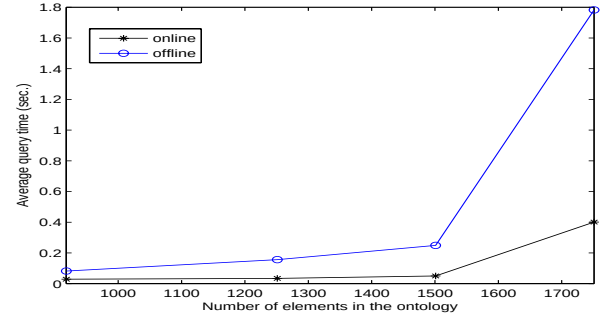


Figure 4: Average query time for different ontology sizes.

experiment are presented in Figure 5. The figure demonstrates that the online mode always outperforms offline mode in terms of average query time.

Load time and reasoner access time may significantly affect the total time required to execute an OLP program in offline and online modes respectively. Therefore, in Figure 6, we demonstrate how load and reasoner access time change as ontology size increases. The figure shows that reasoner access time significantly exceeds the load time and this difference increases as the size of the ontology increases. This means that, execution of OLP programs in online mode takes much more time than their execution in offline mode, especially for large ontologies. We have examined the huge gap between the load time and reasoner access time and revealed that, while an OLP program is being executed in online mode, the reasoner is accessed many times during Prolog’s backtracking. For example, during the execution of the conjunctive query

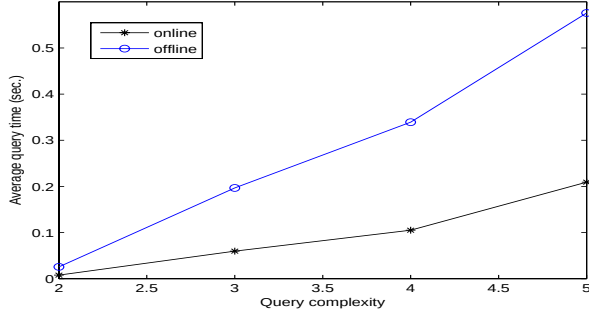


Figure 5: Average query time for different complexities of queries.

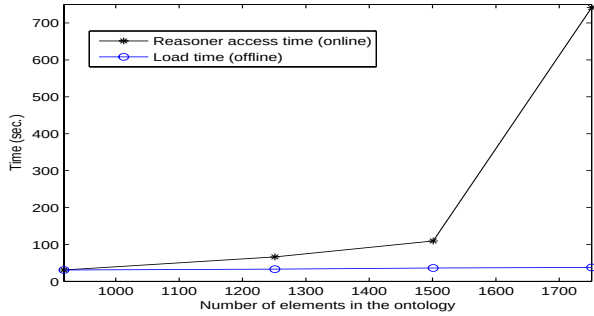


Figure 6: Load time (offline mode) and Reasoner access time (online mode) vs. ontology size.

$q(A1, B1, B2) :- vin: 'Wine'(A1), vin: 'hasWineDescriptor'(A1, B1), not(vin: 'WineSugar'(B1)), vin: 'hasColor'(A1, B2), vin: 'WineColor'(B2)$, the reasoner is accessed 4152 times and 609 solutions are produced at the end of the query execution. Because of the frequent access to the reasoner during backtracking, total time required to execute an OLP program increases significantly in online mode.

During backtracking, the reasoner is accessed repeatedly to retrieve direct or inferred facts about the same set of predicates. Hence, we can use caching mechanisms [8] to store facts returned by the reasoner for a specific ontological predicate and next time we can use these cached facts instead of accessing the reasoner again for the same predicate. With the caching mechanism in place, whenever the OLP interpreter encounters an ontological predicate, first, it checks whether the predicate in concern is already resolved and stored in the cache. If so, it retrieves the appropriate facts from the cache; otherwise, it resolves the predicate by accessing the reasoner and locally caches the retrieved results for future use. Figure 7 shows load time for offline mode and reasoner access time for cache enabled online mode⁵. The results show that online mode with a cache significantly outperforms offline mode. Hence, OLP with a cache can efficiently execute programs with low reasoner access and query time.

4. CASE-STUDY

⁵The cache access time is also included in the reasoner access time.

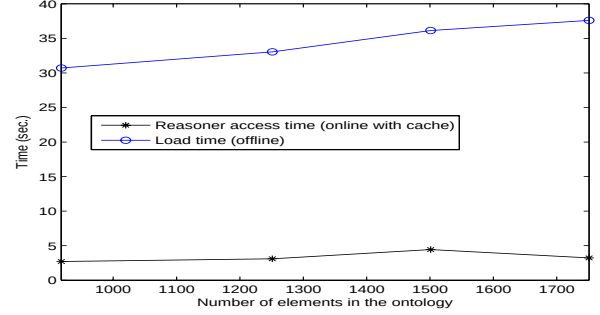


Figure 7: Effect of caching on the performance of OLP online.

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 Alliance⁶ (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)* tasks⁷. In order to solve this problem, we have previously implemented a system called *Sensor Assignment to Missions (SAM)* [9]. Here, we demonstrate how SAM has been significantly improved using OLP.

4.1 ISTAR Tasks and Sensing Resources

We show, in Figure 8, 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 8 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,

⁶http://en.wikipedia.org/wiki/International_Technology_Alliance

⁷<http://en.wikipedia.org/wiki/ISTAR>

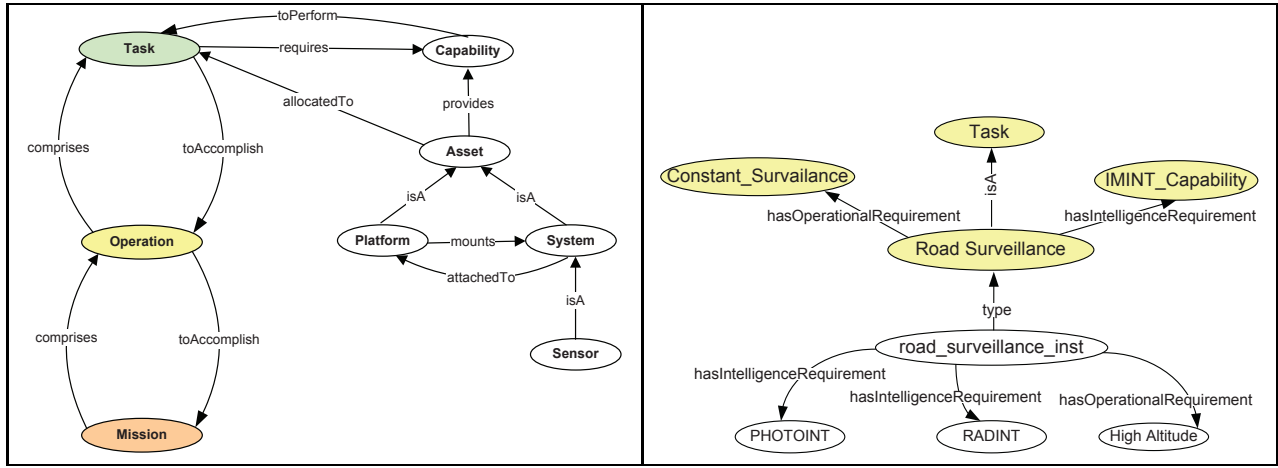


Figure 8: The ISTAR ontology on the left and a task instance example on the right.

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.

4.2 Resource-Task Matchmaking using OLP

The first version of SAM [9] 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. This approach is based on an exhaustive search algorithm and implemented in Java using Pellet for DL reasoning.

Here, we extend SAM via an OLP program shown in Figure 9 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.

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 10 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 9 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

```
%import http://...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).
```

Figure 9: OLP program to compute deployable configurations.

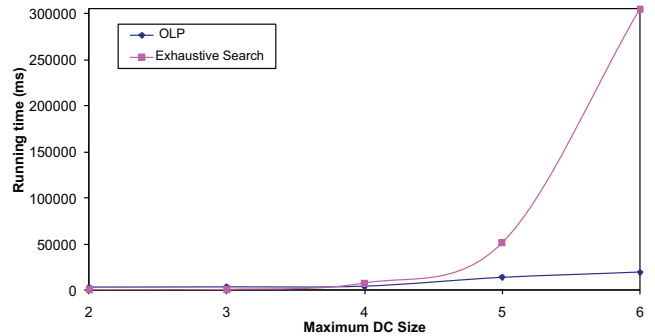


Figure 10: Comparison between OLP program of Figure 9 and the exhaustive search algorithm to find deployable configurations.

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.

5. RELATED WORK AND DISCUSSION

There are various extensions and combinations of logic programming with other programming paradigms. One such combination is functional logic programming [2] merging features of logic and functional programming, efficiently implemented in languages such as Curry [12]⁸, and experiencing a “revival” due to its appeal to Web programming, notably for scripting purposes. Another extension with a potential wide appeal combines logic programming and object-oriented programming [16, 19], making object-oriented programming features such as objects and inheritance available to Prolog programs⁹. Prolog interpreters (e.g., SICStus¹⁰ and Ciao¹¹ Prolog) now commonly allow the seamlessly running, from within a Prolog program, of code implemented in languages such as C or Java. Although these cannot be seen as true extensions, they are very convenient to those wanting to combine functionalities implemented in disparate programming languages.

Rules play an important role in capturing and modeling important domain concepts. Therefore, a lot of effort has been made to develop rule languages and engines for reasoning on top of OWL ontologies. For example, SWRL enables Horn-like rules to be combined with an OWL knowledge base [1]. SWRL aims at extending OWL-DL with semantic rules. Although SWRL does not support negation-as-failure, it implicitly supports classical negation through OWL-DL using classes complements.

Jess [13] is a Java-based expert system shell that uses a RETE algorithm [7] for its forward chaining rule reasoning engine. Jess uses a Common LISP (CLISP) type syntax to describe rules and facts. JessTab [6] is a bridge between Protege [17] and Jess. It enables Jess programs to use and manipulate the knowledge from Protege knowledge bases. This is achieved by mapping Protege knowledge bases to Jess assertions. Originally, JessTab was developed to support Protege-Frames. Thus, JessTab includes only a limited support for handling OWL ontologies. For example, it does not support OWL restrictions and class expressions such as *someValuesFrom* restrictions while mapping OWL ontologies to Jess assertions. In addition to JessTab, there are some other RETE-based rule engines proposed to work with ontologies. Bossam [14] is one of these rule engines. It supports both negation-as-failure and classical negation. It translates OWL documents into built-in list constructs. Then, the reasoning is made based on these construct using a RETE algorithm. SweetJess [10] is a defeasible reasoning system based on the Jess expert system shell. Although it supports the Situated Courteous Logic Programs extension of RuleML, it is restricted to simple terms (variables and atoms).

There are some other approaches based on Prolog. SweetProlog [15] is a Java-based system for translating rules into Prolog. It translates OWL ontologies and rules expressed in RuleML into a set of facts and rules in Prolog. Then, the reasoning about these facts and rules are made completely in Prolog. This approach uses JIProlog as a rule engine. Hence, it translates a OWL subset into simple Prolog predicates which a JIProlog engine can handle. The

main limitation of SweetProlog is its expressive power as it uses Description Logic Programs (DLP) to enable the integration between ontology and rules. DLP is the intersection of DL and Horn logic programs, so it is less expressive than both DL and Horn logic programs. DR-Prolog [5] is a simple rule-based approach to reasoning with incomplete and inconsistent information. It is compatible with RuleML. It is based on the translation of ontological knowledge into Prolog. The system can reason with rules and ontological knowledge written in RDF Schema (RDFS) or OWL. This is achieved through the transformation of the RDFS constructs and many OWL constructs into rules. Note, however, that a number of OWL constructs cannot be captured. SWORIER [20] is a system that uses Prolog to reason about ontologies and rules in order to answer queries. It translates OWL-DL ontologies with rules in SWRL into Prolog using XSLTs (Extensible Stylesheet Language Transformations). Then, query answering is done in Prolog using this translation. It supports only a subset of OWL-DL constructs.

In the approaches described above, ontological knowledge with rules is translated or mapped to Jess or Prolog assertions. On the other hand, OLP keeps ontological knowledge separated from Prolog programs and transparently delegates ontological reasoning to specialised DL reasoners such as Pellet. Hence, it can use the full power of Prolog and the existing reasoners without any loss in the ontological knowledge and expressiveness.

6. 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. We have empirically shown that the frequent access to the reasoner during backtracking is a performance bottle neck for the proposed approach and we have solved this problem using caching. Lastly, using a case-study, we have demonstrated how the proposed approach can be used to solve real-life problems in an efficient and practical way. As future work, we plan to analyse the computational properties of the proposed approach in more challenging settings and demonstrate its usefulness in a variety of real-life problems.

7. REFERENCES

- [1] SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004. <http://www.w3.org/Submission/SWRL>.
- [2] S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, 2010.
- [3] K. R. Apt. *From logic programming to Prolog*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [4] 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.
- [5] A. Bikakis and G. Antoniou. Dr-prolog: a system for reasoning with rules and ontologies on the semantic web. In *AAAI’05: Proceedings of the 20th national conference on Artificial intelligence*, pages 1594–1595. AAAI Press, 2005.
- [6] H. Eriksson. Using jessstab to integrate protégé and jess. *IEEE Intelligent Systems*, 18(2):43–50, 2003.
- [7] C. L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. pages 324–341, 1990.

⁸<http://www-ps.informatik.uni-kiel.de/currywiki/>

⁹A commercial-standard object-oriented logic programming toolkit aptly named Prolog++ can be found at http://www.lpa.co.uk/ppp_det.htm

¹⁰<http://www.sics.se/isl/sicstuswww/site/index.html>

¹¹<http://clip.dia.fi.upm.es/Software/Ciao>

- [8] P. Godfrey and J. Gryz. Answering queries by semantic caches. In *DEXA '99: Proceedings of the 10th International Conference on Database and Expert Systems Applications*, pages 485–498, 1999.
- [9] 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.
- [10] B. N. Grosz, M. D. Gandhe, and T. W. Finin. SweetJess: Translating DAMLRuleML to JESS. In *Proceedings of the International Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML'02)*, 2002.
- [11] 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.
- [12] M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.
- [13] E. F. Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA, 2003.
- [14] M. Jang and J.-C. Sohn. Bossam: An extended rule engine for owl inferencing. In *Proceedings of the 3rd International Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML'04)*, pages 128–138, 2004.
- [15] L. Laera, V. A. M. Tamma, T. J. M. Bench-Capon, and G. Semeraro. Sweetprolog: A system to integrate ontologies and rules. In *Proceedings of the 3rd International Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML'04)*, pages 188–193, 2004.
- [16] C. Moss. *Prolog++: The Power of Object-Oriented and Logic Programming*. Addison-Wesley, 1994.
- [17] N. F. Noy, R. W. Fergerson, and M. A. Musen. The knowledge model of protégé-2000: Combining interoperability and flexibility. In *EKAW '00: Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management*, pages 17–32, London, UK, 2000. Springer-Verlag.
- [18] 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.
- [19] G. Piancastelli, A. Benini, A. Omicini, and A. Ricci. The architecture and design of a malleable object-oriented Prolog engine. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 191–197, New York, NY, USA, 2008. ACM.
- [20] K. Samuel, L. Obrst, S. Stoutenberg, K. Fox, P. Franklin, A. Johnson, K. Laskey, D. Nichols, S. Lopez, and J. Peterson. Translating owl and semantic web rules into prolog: Moving toward description logic programs. *Theory Pract. Log. Program.*, 8(3):301–322, 2008.
- [21] 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.
- [22] M. K. Smith, C. Welty, and D. L. McGuinness. OWL: Web ontology language guide, February 2004.
- [23] L. Sterling and E. Shapiro. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1986.