International Conference on Logic Programming ICLP 2007

Porto, Portugal

8-13 September 2007

ICLP'07 Workshop

ALPSWS2007: Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services

September 13th, 2006

Proceedings

Editors:

S. Heymans, D. Pearce, A. Polleres, E. Ruckhaus, and G. Gupta

Preface

The advent of the Semantic Web promises machine readable semantics and a machine-processable next Generation of the Web. The first step in this direction is the annotation of static data on the Web by machine processable information about knowledge and its structure by means of Ontologies. The next step in this direction is the annotation of dynamic applications and services invocable over the Web in order to facilitate automation of discovery, selection and composition of semantically described services and data sources on the Web by intelligent methods, which is called Semantic Web Services.

This volume contains the papers presented at the second international workshop on *Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS2007)* held on September 13th, 2007 in Porto, Portugal as part of the 23$^{\text{nd}}$ International Conference on Logic Programming (ICLP07).

Many previous workshops and conferences were dedicated to these promising areas mostly with generic topics. With the ALPSWS2007 workshop we have a slighlty different goal. Rather that bringing together people from a widespread variety of research fields with different understandings of the topic we wanted to focus on the various applications areas and approaches in this area from declarative logic programming (LP).

The idea was to get a snapshot of the state of the work related to applications of LP to Semantic Web and Semantic Web Services with the following main objective major benefits:

- Bringing together people from different sub-disciplines of LP and focus on technological solutions and applications from LP to the problems of the Web.
- Promoting further research in this interesting application field.

The ALPSWS 2007 edition focused around *Query Languages on the Web*, *Integrating Ontologies and Rules*, and the more general *Logic Programming on the Semantic Web*, thus conforming what is currently seen as challenges in Web reasoning in general.

September 2007.
Gopal Gupta, Stijn Heymans, Axel Polleres, David Pearce and Edna Ruckhaus

Workshop Organization

Organizing Committee

Stijn Heymans
David Pearce
Axel Polleres
Edna Ruckhaus

Programme Chairs

Gopal Gupta

Programme Committee

Stefan Decker
Pascal Hitzler
Giovambattista Ianni
Zoe Lacroix
Gergely LukÃcsy
Enrico Pontelli
Roman Schindlauer
Hans Tompits
Alejandro Vaisman
Maria-Esther Vidal
Gerd Wagner
Jos de Bruijn

External Reviewers

Francesco Ricca

Table of Contents

ASP — PROLOG: Composition and Interoperation of Rules

C. Baral[1], E. Pontelli[2], and T.C. Son[2]

[1] Arizona State University
Department of Computer Science
chitta@asu.edu
[2] New Mexico State University
Department of Computer Science
{epontell,tson}@cs.nmsu.edu

1 Introduction

One of the main goals of the Semantic Web initiative [3] is to extend the current Web technology to allow for the development of intelligent agents, which can *automatically* and *unambiguously* process the information available on millions of web pages. It has been recognized very early in the development of the Semantic Web that rules are essential for the Web[3] and for Semantic Web applications—e.g., description of semantic web services, rules interchange for e-business applications.

The RuleML initiative is a response to the need of a shared rule markup language using XML markup, which has a precisely defined semantics and efficient implementations. In recent years, a significant amount of work has been devoted to develop knowledge representation languages suitable for the task and a variety of languages for rule markup has been proposed. The initial design [4] included a distinction (in terms of distinct DTDs) between *reaction rules* and *derivation rules*. The first type of rules is used for the encoding of event-condition-action (ECA) rules while the second is meant for the encoding of implicational/inference rules. Despite the fact that many different proposals for ECA rules encoding have appeared the work on ECA rules is still very vague. The most recent modularized description of RuleML [6] reports this area (indicated as *PR RuleML* in that document) as work in progress.

The derivation rules component of the RuleML initiative has originated a family of languages.[4], Datalog plays the role of a core language, with simplified versions (unary and binary Datalog) developed for combining RuleML with OWL (as in SWRL). Various sublanguages have been created to include features like explicit equality (e.g., `fologeq`), negation as failure (e.g., `naffolog`), and Hilog layers (e.g., `hohornlog`). Various authors [7] have argued that any realistic architecture for the Semantic Web must be based on various independent but interoperable languages, including logic programming languages with and without negation-as-failure. The need for these languages and their interaction have been discussed (e.g., [8, 7]). It is also of

[3] `http://www.w3.org/DesignIssues/Rules.html`
[4] `www.ruleml.org/modularization/ruleml_m12n_089_uml_05-06-01.png`.

note that many of the sublanguages of RuleML have been implemented either through translators (e.g., GEDCOM, which translates to XSB and JESS) or engines (e.g., j-DREW, a top-down engine for RuleML, DR-Device, an engine supporting defeasible logic and both strong and default negation, and CommonRules, a bottom-up engine for the Datalog sublanguage).

In this work, we propose a general framework to address the problem of *(i)* inter-operation between knowledge bases encoded using different RuleML languages, and *(ii)* development and integration of different components that reason about RuleML knowledge bases. The approach adopted in this work relies on using a core logic programming framework to address the issues of integration and inter-operation. In particular, the spirit of our approach relies on the following beliefs:

- the natural semantics of various levels of the RuleML deduction rules hierarchy can be captured by different flavors of logic programming;
- modern logic programming systems are provided with foreign interfaces that allow declarative interfacing to other paradigms.

The idea is to combine the ASP-Prolog framework of [5] with the notations for modularization of answer set programming of [2]. The result is a logic programming framework, where modules responding to different logic programming semantics (e.g., Herbrand minimal model, well-founded semantics, answer set semantics) can co-exist and interoperate.

The framework provides a natural answer to the problems of use and inter-operation of RuleML knowledge. Most of the emphasis is on using answer set programming to handle some of the sublanguages (e.g., datalog, ur-datalog, nafdatalog and negdatalog), though the core framework will naturally support most of the languages (e.g., hornlog, hohornlog).

A thorough presentation of the framework can be found in [1].

References

[1] C. Baral et al. A Framework for Composition and Interoperation of Rules in the Semantic Web. *RuleML*, Springer Verlag, pp. 39-50, 2006.

[2] C. Baral et al. Macros, macro calls, and use of ensembles in modular answer set programming. *ICLP*, Springer, 2006.

[3] T. Berners-Lee et al. The semantic web. In Scientific American, May 2001.

[4] H. Boley et al. RuleML Design, Version 0.8. 2002-09-03, 2002.

[5] O. Elkhatib et al. A Tool for Knowledge Base Integration and Querying. *AAAI Spring Symp.*, AAAI Press, 2006.

[6] D. Hirtle, H. Boley. The Modularization of RuleML. 2005-12-15, 2005.

[7] M. Kifer et al. A Realistic Architecture for the Semantic Web. *RuleML*, Springer Verlag, pp. 17–29, 2005.

[8] W. May et al. Active Rules in the Semantic Web: Dealing with Language Heterogeneity. *RuleML*, Springer, 2005.

dlvhex-sparql: A SPARQL-compliant Query Engine based on dlvhex *

Axel Polleres[1] and Roman Schindlauer[2]

[1] DERI Galway, National University of Ireland, Galway
axel@polleres.net
[2] Univ. della Calabria, Rende, Italy and Vienna Univ. of Technology, Austria
roman@kr.tuwien.ac.at

Abstract. This paper describes the dlvhex SPARQL plugin, a query processor for the upcoming Semantic Web query language standard by W3C. We report on the implementation of this languages using dlvhex, a flexible plugin system on top of the DLV solver. This work advances our earlier translation based on the semantics by Perez et al. towards an engine which is fully compliant to the official SPARQL specification. As it turns out, the differences between these two definitions of SPARQL, which might seem moderate at first glance, need some extra machinery. We also briefly report the status of implementation, and extensions currently being implemented, such as handling of aggregates, nested CONSTRUCT queries in the spirit of networked RDF graphs, or partially support of RDFS entailment. For such extensions a tight integration of SPARQL query processing and Answer-Set Programming, the underlying logic programming formalism of our engine, turns out to be particularly useful, as the resulting programs can actually involve unstratified negation.

1 Introduction

SPARQL, the upcoming Semantic Web query language, is short before being standardized by the W3C, and has just reached Candidate Recommendation Status [8]. As opposed to earlier versions of this specification, the formal underpinnings of the language have been seriously improved, influenced by results from academia such as Perez et al.'s work [6]. In [7] we presented a translation from SPARQL to Datalog following Perez et al. and showed how we can cover even corner-cases such as non-well-designed query patterns, where UNION and OPTIONAL patterns turned out to be particularly tricky. In the present work we aim at bridging the gap between the formal translation from [7] towards an actual implementation of the official W3C candidate recommendation. Compared with the semantics presented in [6, 7], the recent specification shows some differences which require additional machinery, such as the treatment of filters in optional graph patterns, multiset semantics, and the handling of blank nodes in CONSTRUCT queries which have an impact for practical implementations. This paper is to

* This work has been supported by the European FP6 projects inContext (IST-034718) and REWERSE (IST 506779), by the Austrian Science Fund (FWF) project P17212-N04, by the AECI Programa de Cooperación Interuniversitaria e Investigación Científica entre España y los pases de Iberoamérica (PCI), by the Consejería de Educación de la Comunidad de Madrid and Universidad Rey Juan Carlos under the project URJC-CM-2006-CET-0300, as well as by Science Foundation Ireland under the Lion project (SFI/02/CE1/I131).

be conceived as a system description: Rather by use of practical examples than repeating formal details from our earlier works, we will show how our earlier translation can be lifted to a more spec-compliant one. Moreover, we report on implementation details of our prototypical engine dlvhex-sparql.

We will review the basics of dlvhex and main ideas of our translation by means of simple examples in Section 2. In Subsection 2.3 we will discuss the main differences between our original semantics from [7] and the current SPARQL specification [8] along with patches for our translation. Next, we will present some details about our prototype implementation in Section 3. Finally, in Section 4 we will motivate further why a tight integration of SPARQL query processing and answer-set programming, the underlying logic programming of our engine, turns out to be particularly useful and will give an outlook to future work.

2 From SPARQL to dlvhex

As shown in [7] the semantics of SPARQL SELECT queries can, to a large extent, be translated to Datalog programs with minimal support of built-in predicates. Hence, any logic programming engine which supports Datalog (i.e., function-free) with negation as failure, as well as built-in functions to import triples from given RDF graphs, could in principle serve as a SPARQL engine. We will focus here particularly on our implementation of this translation using the dlvhex engine, a flexible and extensible plugin framework on top of the DLV system to support a wide range of external predicates.

2.1 dlvhex Basics

dlvhex [3] is a reasoner for so-called HEX-programs [11], a relatively new logic programming language, which provides an interface to external sources of knowledge. The definition of this interface is very general, allowing for the implementation of a wide range of specialized tasks, such as the import of RDF data, basic string manipulation routines, or even aggregate functions.

The crucial feature of HEX-programs are *external atoms*, which are of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m),$$

where $Y_1, \ldots, Y_n$ is a list of predicates and terms and $X_1, \ldots, X_m$ is a list of terms (called *input list* and *output list*, respectively), and g and *output arities* n and m fixed for g. Intuitively, an external atom provides a way for deciding the truth value of an output tuple depending on the extension of a set of input predicates and terms. Note that this means that external predicates, unlike usual definitions of built-ins in logic programming, can not only take constant parameters but also (extensions of) predicates as input.

A *rule* is of the form

$$h \ :- \ b_1, \ldots, b_m, not\, b_{m+1}, \ldots not\, b_n \tag{1}$$

where h and b_i ($1 \leq i \leq n$) are atoms, b_k ($1 \leq k \leq m$) are either atoms or external atoms, and 'not' is the symbol for negation as failure.

The semantics of dlvhex generalizes the well-known answer-set semantics [4] by extending it to external atoms. One distinguished feature of the answer-set semantics is

[3] Available on http://www.kr.tuwien.ac.at/research/dlvhex/.

its ability to generate multiple minimal models for a single problem specification. Its treatment of negation as failure qualifies it as an intuitive way to deal with unstratified negation in logic programming. Answer-set programming is particularly suitable for combinatorial search problems and their applications.

In our implementation we translate SPARQL queries to HEX-programs with a set of dedicated external atoms, only two of which we mention here explicitly. Further external atoms and built-in functions are necessary to deal with complex FILTER expressions as defined in [8, Sec. 11.3], where we refer to [7, 9] for further details.

RDF Import The access on RDF knowledge is realized through the `&rdf` predicate. It is of the form $\&rdf[i](s, p, o)$, where both the input term i as well as the output terms s, p, o are constants The external atom $\&rdf[i](s, p, o)$ is true if (s, p, o) is an RDF triple *entailed* by the RDF graph which is accessibly at IRI i. Here, we consider simple RDF entailment [5] only.

Skolemizing Blank Nodes In order to properly deal with blank nodes in CONSTRUCTs (see Subsection 2.3), we need to be able to generate fresh blank node identifiers. The idea here is similar to Skolemization. The external predicate $\&sk[id, v_1, \ldots, v_n](sk_{n+1})$ computes a unique, new "Skolem"-like term $id(v_1, \ldots, v_n)$, from its input parameters.

As widely known for programs without external predicates, safety [12] guarantees that the number of entailed ground atoms is finite. Though, by external atoms in rule bodies, new, possibly infinitely many, ground atoms could be generated, even if all atoms themselves are safe. In order to avoid this, the notion of *strong safety* [11] for HEX-programs, constrains the use of external atoms in cyclic rules and guarantees finiteness of models as well as finite computability of external atoms.

2.2 From SPARQL to dlvhex by Example

In this section, we exemplify our translation by means of some illustrating sample SPARQL queries. We assume basic familiarity of the reader with RDF and SPARQL, and will only briefly intoduce some basics here: We define a SPARQL *query* as a tuple $Q = (R, P, DS)$ where R is a result form, P a graph pattern, and DS a dataset.[4] For a SELECT query, a *result form* R is simply a set of variables, whereas for a CONSTRUCT query, the result form R is a set of triple patterns.

We assume the pairwise disjoint, infinite sets I, B, L and Var, which denote IRIs, blank node identifiers, RDF literals,[5] and variables respectively.
Graph patterns are recursively defined as follows:

- $s\ p\ o.$ is a graph pattern where $s, o \in I \cup B \cup L \cup Var$ and $p \in I \cup Var$.
- A set of graph patterns is a graph pattern.
- Let P, P_1, P_2 be graph patterns, R a filter expression, and $i \in I \cup Var$, then P_1 OPTIONAL P_2,P_1 UNION P_2,GRAPH i P, and P FILTER R are graph patterns.

For any pattern P, we denote by $vars(P)$ the set of all variables occurring in P and by $\overline{vars}(P)$ the tuple obtained by the lexicographic ordering of all variables in P. As *atomic filter expression*, we allow here the unary predicates BOUND (possibly with

[4] We will ignore solution modifiers for the purpose of this paper, since they can be added by post-processing results of our translation.

[5] For sake of brevity, we only cover plain (i.e., untyped, not language tagged) literals here.

variables as arguments), isBLANK, isIRI, isLITERAL, and binary comparison predicates '$=$', '$<$', '$>$' with arbitrary safe built-in terms as arguments. *Complex filter expressions* can be built using the connectives '$\neg$', '$\wedge$', and '$\vee$'.

The *dataset* $DS = (G, \{(g_1, G_1), \ldots, (g_k, G_k)\})$ of a SPARQL query is defined by a default graph G plus a set of named graphs, i.e., pairs of IRIs and corresponding graphs. Without loss of generality (there are other ways to define the dataset such as in a SPARQL protocol query), we assume G given as the merge of the graphs denoted by the IRIs explicitly given in a set of FROM clauses and the named graphs $g_1, \ldots, g_k$ are specified in the form of FROM NAMED clauses.

For the following example queries, we assume the datasets consisting of two RDF graphs with the URIs `http://alice.org` and `http://ex.org/bob` which contain some information about Alice and Bob encoded in the commonly used FOAF[6] vocabulary. For instance, the following SELECT query $Q = (R, P, DS)$ selects all persons who know somebody, and the names of these persons.[7]

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . ?X foaf:knows _:x . }
```

Here, P is a simple set of triple patterns, also called *basic graph pattern* in SPARQL. R is the set of variables $\{?X, ?Y\}$, $DS = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$, i.e., the default graph being the merge of the two graphs and an empty set of named graphs, since no FROM NAMED clause is given. Using the RDF-plugin of dlvhex such queries can be translated to a simple HEX-program:[8]

```
(1)  tripleQ(S,P,O,def)   :-  &rdf["http://ex.org/bob"](S,P,O).
(2)  tripleQ(S,P,O,def)   :-  &rdf["http://alice.org"](S,P,O).
(3)  answerP(X,Y,BLANK_x,DS)  :-  triple(X,rdf:type,foaf:Person,DS),
                                  triple(X,foaf:name,Y,DS),
                                  triple(X,foaf:knows,BLANK_x,DS).
(4)  answerQ(X,Y)   :-  answerP(X,Y,BLANK_x,def).
```

Here, rules (1)+(2) "collect" the dataset by merging the two source graphs in the predicate $\texttt{triple}_Q$, where the constant `def` in last parameter denotes the triples of the default graph. Disambiguation of possible overlapping blank node ids in the source graphs is taken care of by the RDF plugin, i.e., during import the `&rdf` predicate gives a fresh id to any blank node. As we can see in rule (3), basic graph patterns basically boil down to simple conjunctive queries over the predicate $\texttt{triple}_Q$ of which the results are collected in the predicate $\texttt{answer}_P(\overline{vars}(P), \texttt{DS})$. The variable `DS` denotes the part of the dataset the pattern refers to (see the next example for more details). Blank nodes in P are simply treated as special variables, which is a quite standard procedure (see e.g., [3].[9] The projection to the variables in R and restriction to results from the default graph takes place in rule (4), which finally collects all solution tuples for the query in the dedicated predicate $\texttt{answer}_Q$.

[6] `http://xmlns.com/foaf/spec/`

[7] As usual in SPARQL or Turtle [2], the predicate '`rdf:type`' is abbreviated with '`a`'.

[8] dlvhex uses the common PROLOG style notation where unquoted uppercase terms denote variables and all other terms denote constants. We simplify here from the notation used in the actual implementation, where some encoding is necessary in order to distinguish RDF literals, IRIs, blank nodes, etc.

[9] For the treatment of blank nodes in R, i.e., in CONSTRUCTs, we refer to Section 2.3 below.

More complex, possibly nested patterns are handled by introducing, for each subpattern of P, new auxiliary predicates $\texttt{answer1}_P$, $\texttt{answer2}_P$, $\texttt{answer3}_P$, etc. We exemplify this by the following GRAPH query which selects creators of graphs and the persons they know.

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM NAMED <http://alice.org>
FROM NAMED <http://ex.org/bob>
WHERE { ?G foaf:maker ?X .
        GRAPH ?G { ?X foaf:knows ?Y . ?Y a foaf:Person . } }
```

This query is translated to a HEX-program as follows:

```
(1) tripleQ(S,P,O,def)  :-  &rdf["http://alice.org"](S,P,O).
(2) tripleQ(S,P,O,"http://alice.org")  :-  &rdf["http://alice.org"](S,P,O).
(3) tripleQ(S,P,O,"http://ex.org/bob")  :-  &rdf["http://ex.org/bob"](S,P,O).
(4) answer1P(X,Y,DS)  :-  tripleQ(G,foaf:maker,X,DS), answer2(X,Y,G), G != def.
(5) answer2P(X,Y,DS)  :-  tripleQ(X,foaf:knows,Y,DS),
                          tripleQ(X,rdf:type,foaf:Person,DS).
(6) answerQ(X,Y)  :-  answer1P(X,Y,def).
```

Here, again the first rules (1)-(3) import the dataset, now also involving named graphs. The GRAPH subpattern is computed by predicate $\texttt{answer2}_P$, and we see that the last parameter in the triple predicate carries over bindings to particular named graphs or via the constant `def` to the default graph. Note that the inequality atom `G != def` in rule (4) serves to restrict answers for the GRAPH subpattern to only refer to named graphs, according to SPARQL's semantics.

Next, let us turn to a query that involves a UNION pattern, asking for persons and their names *or* nicknames.

```
SELECT ?X ?Y ?Z FROM ...
WHERE { ?X a foaf:Person { { ?X foaf:name ?Y. } UNION { ?X foaf:nick ?Z. } } }
```

Alternatives can be modeled by splitting off the branches in a UNION pattern into several rules with the same `answer` head predicate:

```
(1) tripleQ(S,P,O,def)  :-  ...
(2) answer1P(X,Y,Z,DS)  :-  tripleQ(X,rdf:type,foaf:Person,DS), answer2P(X,Y,Z,DS).
(3) answer2P(X,Y,null,DS)  :-  tripleQ(X,foaf:name,Y,DS).
(4) answer2P(X,null,Z,DS)  :-  tripleQ(X,foaf:nick,Z,DS).
(6) answerQ(X,Y,Z)  :-  answer1P(X,Y,Z,def).
```

Since we bind names and nicknames to different variables Y and Z here, the answers for the non-occurring variable will be unbound in the respective branch of the UNION. We emulate such unboundedness in our translation by null values [7] in the rules (3)+(4).

Let us turn to OPTIONAL patterns by the following example query which selects all persons and optionally their names:

```
SELECT * WHERE { ?X a foaf:Person.  OPTIONAL { ?X foaf:name ?N } }
```

OPTIONALs can be emulated again by null values and using negation as failure.

```
(1) tripleQ(S,P,O,def)  :-  ...
(2) answer1P(N,X,DS)  :-  tripleQ(X,rdf:type,foaf:Person,DS), answer2P(N,X,DS).
(3) answer1P(null,X,DS)  :-  tripleQ(X,rdf:type,foaf:Person,DS),
                             not answer2'P(X,DS).
(4) answer2P(N,X,DS)  :-  tripleQ(X,foaf:name,N,DS).
(5) answer2'P(X,DS)  :-  answer2P(N,X,DS).
(6) answerQ(N,X)  :-  answer1P(N,X,def).
```

In rules (3)+(5) we cover the case where the optional part has no solutions for X by a rule with head predicate $\texttt{answer2'}_P$ which projects away all variables only occurring in the optional part ($\texttt{answer2}_P$) and which we negate in rule (3).

As for the treatment of FILTER expressions, we made the restricting assumption in [7] that each variable appearing in a FILTER expression needs to be bound in a triple pattern in the same scope as the FILTER expression, since otherwise our recursive translation given in [7] would construct possible unsafe rules. Take for instance the pattern $P = \{$ `?X foaf:mbox ?M . FILTER(?Age > 30)` $\}$. In our translation without further modification, such a pattern this would yield a rule like:

$$\texttt{answer}_P\texttt{(Age,M,X,DS)} \texttt{ :- } \texttt{triple}_Q\texttt{(X,foaf:mbox,M,DS), Age > 30.}$$

where `Age` is an unsafe variable occurring in a built-in atom. however, it turns out that the safety restriction for variables in FILTERs is unnecessary, since we could remedy the situation by replacing all unsafe variables in a FILTER simply by the constant null again, which yields for our example pattern P:

$$\texttt{answer}_P\texttt{(null,M,X,DS)} \texttt{ :- } \texttt{triple}_Q\texttt{(X,foaf:mbox,M,DS), null > 30.}$$

As expected, this rule can never fire, since the built-in atom `null > 30` is always false.

Finally, let us turn our attention to CONSTRUCT queries. We suggested in [7] that we can allow CONSTRUCT queries of the form $Q = (R, P, DS)$ where R consists of bNode-free triple patterns. We can model these by adding a rule

$$\texttt{triple}_Q\texttt{(}s,p,o\texttt{,res)} \texttt{ :- } \texttt{answer}_Q\texttt{(}\overline{vars}(P)\texttt{)}.$$

for each triple pattern $s\ p\ o.$ in R[10] to the translated program. The result graph `res` is then naturally represented in the answer set of the extended program, namely by those tuples in the extension of the predicate $\texttt{triple}_Q$ having `res` as the last parameter and representing valid RDF triples.

Apart from some extra-machinery, which is needed in the case of non-well-designed graph patterns [6], the examples in this section should cover the basic ideas behind our translation which had been described in [7], and we refer the interested reader to this work for further details. The present paper is rather focused on implementation specific aspects concerning the latest official specification of SPARQL and some difficulties which arise from particular decisions taken by the W3C Data Access working group. We will cover these issues in the next subsection.

2.3 Full SPARQL Compliance

In order to arrive at a SPARQL compliant translation, we face the following difficulties:

1. How to deal with solution modifiers.
2. SPARQL defines a multi-set semantics.
3. SPARQL allows FILTER expressions in OPTIONAL patterns to refer to variables bound outside the enclosing OPTIONAL pattern.
4. SPARQL allows blank nodes in the result form of CONSTRUCT queries.

[10] Analogously to the FILTER example, we can replace variables unbound in P but occurring in R by null again in order to ensure safety.

As for 1, we do not yet treat solution modifiers such as ORDER BY and OFFSET in our current prototype, but these can be easily added by post-processing the results obtained from our translation fed into dlvhex. Issue 2 is somewhat harder to solve. Note that our current translation, as well as the SPARQL semantics defined by Perez et al. [6] creates sets of solutions, i.e., each query is treated as if it was a DISTINCT query. Take for instance a variation of our UNION example from above:

```
SELECT ?N FROM ... WHERE { { ?X foaf:name ?N. } UNION { ?X foaf:nick ?N. } }
```

and assume the source graph

```
:bob foaf:name "Bob" ; foaf:nick "Bobby" .
:alice foaf:knows _:a .
_:a foaf:name "Bob"; foaf:nick "Bob"; foaf:nick "Bobby" .
```

The naive translation of the above query to a HEX-program is as follows:

```
(1) tripleQ(S,P,O,def)  :-  ...
(2) answer1P(N,X,DS)  :-  tripleQ(X,foaf:name,N,DS).
(3) answer1P(N,X,DS)  :-  tripleQ(X,foaf:nick,N,DS).
(4) answerQ(N)  :-  answer1P(N,X,def).
```

This program (in a bottom-up evaluation such as the one underlying the dlvhex system) would result in two answers $\texttt{answer}_Q\texttt{("Bob")}$ and $\texttt{answer}_Q\texttt{("Bobby")}$. According to the official SPARQL semantics, however, the above query has four solutions binding variable `?N` three times to `"Bob"` and twice to `"Bobby"`. If we observe where the duplicates get "lost" in our translation, we can see that only (i) the final projection in predicate $\texttt{answer}_Q$ and (ii) duplicates due to UNION patterns cause us to lose duplicates. We can remedy this easily by (i) always carrying over all the variables in all subpatterns to the $\texttt{answer}_Q$ predicate and only projecting out the non-selected variables during postprocessing, and (ii) adding an extra variable for each UNION pattern which models possible branches a solution stems from. The such modified version of our translated program looks as follows:

```
(1)  tripleQ(S,P,O,def)  :-  ...
(2') answer1P(N,X,1,DS)  :-  tripleQ(X,foaf:name,N,DS).
(3') answer1P(N,X,2,DS)  :-  tripleQ(X,foaf:nick,N,DS).
(4') answerQ(N,X,Union1)  :-  answer1P(X,N,Union1,def).
```

Here, the constants `1` and `2` mark the branches of the union in rules (2')+(3'), and are carried over to the end result in rule (4') by the extra variable `Union1`. Indeed, this modified program has four answers $\texttt{answer}_Q\texttt{("Bob",:bob,1)}$, $\texttt{answer}_Q\texttt{("Bobby",:bob,2)}$, $\texttt{answer}_Q\texttt{("Bobby",\_:a,2)}$, and $\texttt{answer}_Q\texttt{("Bob",\_:a,2)}$.

Regarding issue 3, let us consider a query involving the above mentioned FILTER condition:

```
SELECT ?N ?M WHERE { ?X foaf:name ?N . ?X :age ?Age .
                     OPTIONAL { ?X foaf:mbox ?M . FILTER(?Age > 30) } }
```

Here, we want to select names of persons and only output email addresses (`foaf:mbox`) of those ones older than 30. The possibility of FILTERs within OPTIONALs to refer to variables bound outside the enclosing OPTIONAL pattern is an interesting feature of SPARQL for such queries, however, our original translation would treat filters strictly local to their pattern:

```
(1)  triple_Q(S,P,O,def)  :-  ...
(2)  answer1_P(Age,N,M,X,DS)  :-  triple_Q(X,foaf:name,N,DS), triple_Q(X,:age,Age,DS),
                                  answer2_P(Age,M,X,DS).
(3)  answer1_P(Age,N,null,X,DS)  :-  triple_Q(X,foaf:name,N,DS),
                                     triple_Q(X,:age,Age,DS),
                                     not answer2'_P(Age,X,DS).
(4)  answer2_P(null,M,X,DS)  :-  triple_Q(X,foaf:mbox,M,DS), null > 30.
(5)  answer2'_P(Age,X,DS)  :-  answer2_P(Age,M,X,DS).
(6)  answer_Q(N,M)  :-  answer1_P(Age,N,M,X,def).
```

Since the use of variable `Age` in rule (4) would be unsafe, our original translation replaces it by null, thus not returning any email addresses (i.e., bindings for `N`) for the overall query. The solution is now to modify the translation in order to draw FILTERs in the scope of OPTIONALs upwards in the pattern tree, yielding a modified translation:

```
(1)  triple_Q(S,P,O,def)  :-  ...
(2') answer1_P(Age,N,M,X,DS)  :-  triple_Q(X,foaf:name,N,DS), triple_Q(X,:age,Age,DS),
                                  answer2_P(M,X,DS), Age > 30.
(3a') answer1_P(Age,N,null,X,DS)  :-  triple_Q(X,foaf:name,N,DS),
                                      triple_Q(X,:age,Age,DS),
                                      answer2_P(M,X,DS), not Age > 30.
(3b') answer1_P(Age,N,null,X,DS)  :-  triple_Q(X,foaf:name,N,DS),
                                      triple_Q(X,:age,Age,DS), not answer2'_P(X,DS).
(4') answer2_P(M,X,DS)  :-  triple_Q(X,foaf:mbox,M,DS).
(5') answer2'_P(X,DS)  :-  answer2_P(M,X,DS).
(6') answer_Q(N,M)  :-  answer1_P(Age,N,M,X,def).
```

Rules (2')-(3b') now exactly reflect the case distinction for OPTIONALs by the definition of the LeftJoin operator in [8, Section 12.4]. As an interesting side-note, we remark that the non-local behavior of filter expressions only applies to FILTERs on the top level of OPTIONALs: The reader might easily convince herself by the definitions in the current SPARQL specification that a slightly modified query

```
SELECT ?N ?M WHERE { ?X foaf:name ?N . ?X :age ?Age .
             OPTIONAL {?X foaf:name ?N { ?X foaf:mbox ?M . FILTER(?Age > 30) } } }
```

is not semantically equivalent to the original query although the triple `?X foaf:name ?N` inside the OPTIONAL seems to be redundant at first glance. In fact, the difference here is that FILTERs which are nested within a group graph pattern will be evaluated local to this pattern, not taking bindings from outside the OPTIONAL into account.

Finally, let us turn to issue 4, namely the translation of CONSTRUCT queries involving blank nodes in the result form. We consider an example query which constructs `foaf:maker` relations for people authoring a document, expressed by the Dublin Core property `dc:creator`. We assume that in the source graph all values for `dc:creator` are literals denoting the authors' names. Thus, we want to create bNodes for each author, since the `foaf:maker` of a document should be a `foaf:Agent`:

```
CONSTRUCT { _:b a foaf:Agent. _:b foaf:name ?N. ?Doc foaf:maker _:b. } FROM ...
    WHERE { ?Doc dc:creator ?N. }
```

The idea to implement the SPARQL semantics properly here is to use the external predicate `&sk` mentioned in Subsection 2.1 to generate new blank node identifiers for each solution binding for $\overline{var}(P)$ similar in spirit to Skolemization . We simply use the original bNode identifier `b` in R as "Skolem function":

```
(1) triple_Res(S,P,O,def)  :-  ...
(2) answer1_P(Doc,N,DS)  :-  triple_Q(Doc,dc:creator,N,DS).
(3) triple_Res(BLANK_b,rdf:type,foaf:Agent,res)  :-  answer1_P(Doc,N,def),
                                                     &sk[b,Doc,N](BLANK_b).
(4) triple_Res(BLANK_b,foaf:name,N,res)  :-  answer1_P(Doc,N,def),
                                             &sk[b,Doc,N](BLANK_b).
(5) triple_Res(Doc,foaf:maker,BLANK_b,res)  :-  answer1_P(Doc,N,def),
                                                &sk[b,Doc,N](BLANK_b).
```

Note that, since we use different predicates $\texttt{triple}_{Res}$ and $\texttt{triple}_{Q}$for the result triples and dataset triples here, the resulting program stays in principle non-recursive and thus strong safety as discussed in Subsection 2.1 is guaranteed, despite the generation of new values by means of the external predicate `&sk`.

3 Prototype Implementation

We implemented a prototype of a SPARQL engine based on the dlvhex solver, called dlvhex-sparql. The external atoms in HEX-programs are provided by so-called *plugins*, which are dynamically loaded at run-time by the evaluation framework of dlvhex. A plugin may also supply a rewriting module, which is executed prior to the model generation algorithm and allows for a conversion of the input data into a valid HEX-program. The prototype exploits the rewriting mechanism of the dlvhex framework, taking care of the translation of a SPARQL query into the appropriate HEX-program, as laid out in Subsection 2.2. The system implements external atoms used in the translation, namely (i) the `&rdf`-atom for data import aggregate atoms, and (ii) a string manipulation atom implementing the `&sk`-atom for blank node handling. The default syntax of a dlvhex results corresponds to the usual answer format of logic programming engines, i.e., sets of facts, from which we generate an XML representation that can subsequently be transformed easily to a valid RDF syntax by an XSLT to export solution graphs.

Note that the support of complex FILTER expressions is only rudimentary at the moment and subject to ongoing work. As mentioned before, we will need a dedicated set of additional external atoms in order to support the full extent of FILTER expressions as described in [8, Sec. 11.3].

We also implemented a rudimentary Web service interface making our engine accessible as a general purpose SPARQL endpoint. This was realized by XSL transforming the XML output of dlvhex into the result format prescribed by the SPARQL protocol[11] and is accessible via a SOAP interface at `http://apolleres.escet.urjc.es:8080/axis/services/SparqlEvaluator?wsdl`.

4 Extensions and Next Steps

While the current implementation efforts around dlvhex-sparql described here were focused on conceptually proving the feasibility of a fully SPARQL compliant query engine on top of dlvhex, our intentions behind go well beyond this sheer exercise. We are currently working on extensions such as allowing aggregate and built-in functions in the result form of queries, which allows computations of new values. Such an extension is crucial for instance for mapping between different, overlapping RDF vocabularies [1]. In this context, we plan to support the use of CONSTRUCT queries as part of the dataset which allows to express such mappings[12] or interlinked, implicit RDF metadata[13]. The embedding of such extensions into our translation comes mostly without

[11] `http://www.w3.org/TR/2006/CR-rdf-sparql-protocol-20060406/`

[12] `www.rdfweb.org/topic/ExpertFinder_2fmappings`

[13] `www.w3.org/2005/rules/wg/wiki/UCR/Publishing_Rules_for_Interlinked_Metadata`

additional costs, since the respective query translations for both the actual query as well as mapping rules and views in the form of CONSTRUCTs can be translated into a single dlvhex program and evaluated at once. Here is where the power of answer-set programming comes into play, since such combined programs may involve unstratified recursion which can be dealt with flexibly under brave or cautions reasoning, respectively. We should mention here related approaches such as [10], which alternatively suggest the use of the well-founded semantics for such scenarios, but with a similar intention to create networks of RDF graphs (possibly recursively) linked by CONSTRUCT queries. Moreover, we did not yet conduct extensive performance evaluations, but we would not expect to be necessarily competitive with special-purpose SPARQL engines. However, the power of our approach lies in its natural combination of RDF with the rules world, which for instance allows us to plug-in on the fly Datalog rulesets which emulate RDF(S) entailment (see for instance[3]).

References

1. B. Aleman-Meza, U. Bojars, H. Boley, J. G. Breslin, M. Mochol, L. J. Nixon, A. Polleres, and A. V. Zhdanova. Combining RDF vocabularies for expert finding. In *Proceedings of the 4th European Semantic Web Conference (ESWC2007)*, number 4519 in Lecture Notes in Computer Science, pages 235–250, Innsbruck, Austria, June 2007. Springer.
2. D. Beckett. Turtle - Terse RDF Triple Language, Apr. 2006. Available at `http://www.dajobe.org/2004/01/turtle/`.
3. J. de Bruijn and S. Heymans. RDF and logic: Reasoning and extension. In *Proceedings of the 6th International Workshop on Web Semantics (WebS 2007), in conjunction with the 18th International Conference on Database and Expert Systems Applications (DEXA 2007)*, Regensburg, Germany, September 3–7 2007. IEEE Computer Society Press.
4. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
5. P. Hayes. RDF semantics. Technical report, W3C, February 2004. W3C Recommendation.
6. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *International Semantic Web Conference (ISWC 2006)*, pages 30–43, 2006.
7. A. Polleres. From SPARQL to rules (and back). In *Proceedings of the 16th World Wide Web Conference (WWW2007)*, Banff, Canada, May 2007.
8. E. Prud'hommeaux and A. Seaborne (eds.). SPARQL Query Language for RDF, June 2007. W3C Candidate Recommendation, available at `http://www.w3.org/TR/2007/CR-rdf-sparql-query-20070614/`.
9. S. Schenk. A SPARQL Semantics Based on Datalog. In *KI2007, Osnabrck, Germany*, 2007.
10. S. Schenk and S. Staab. Networked RDF Graphs. Tech. rep., Univ. Koblenz, 2007. `http://www.uni-koblenz.de/~sschenk/publications/2006/ngtr.pdf`.
11. R. Schindlauer. *Answer-Set Programming for the Semantic Web*. PhD thesis, Vienna University of Technology, Dec. 2006.
12. J. Ullman. *Principles of Database & Knowledge Base Systems*. Comp. Science Press, 1989.

OnEQL: An Ontology Efficient Query Language Engine for the Semantic Web

Edna Ruckhaus and Eduardo Ruiz and María Esther Vidal

Universidad Simón Bolívar
Caracas, Venezuela
{ruckhaus,eruiz, mvidal}@ldc.usb.ve

Abstract. In this paper we describe the OnEQL system, a query engine that implements optimization techniques and evaluation strategies to speed up the evaluation time of querying and reasoning services in the Semantic Web. To identify execution plans that reduce the cost of evaluating a query, we developed a twofold optimization strategy that combines cost-based optimization and Magic Sets techniques. In the first stage, a dynamic programming-based algorithm is used to identify an ordering of predicates in the query that minimizes its estimated evaluation cost. In the second stage, Magic Sets techniques are used to push down query selections into the OnEQL ontology representation, in order to reduce the number of facts inferred during query evaluation. Additionally, we developed three physical operators that execute the sideways passing of bindings during the evaluation of the execution plan. To illustrate the advantages of this approach, we report the results of an experimental study over the most popular health ontologies.

1 Introduction

Ontologies play an important role in the Semantic Web, and provide the basics for the definition of concepts and relationships that make global interoperability possible. Knowledge represented in ontologies can be used to annotate data, distinguish similar concepts, and generalize and specialize concepts.

A great number of ontologies have become available under the umbrella of the Semantic Web. In particular, for the health domain, large ontologies have been defined, for example, MeSH [15], Disease [4], Galen [6], and EHR_RM [5], which are commonly used by the health and bioinformatics community to find solutions for a variety of problems. These ontologies are specified in different standard languages such as XMLSchema [25], OWL [14] or RDFS [2]; and regular requirements are expressed using query languages such as SPARQL [17] or RQL [12]. OWL is a markup language that extends the graph-based model used by RDF and provides complex structures and different levels of complexity on top of XML. OWL is commonly used to share and publish information encoded in an ontology. On the other hand, SPARQL is an RDF-based query language that enables users to select portions of an ontology that satisfy certain patterns or conditions. In the Semantic Web, OWL and SPARQL have become

standards to publish and query data, respectively. In this paper, we propose the OnEQL system which interoperates between different ontology representations and query languages. In the current version of OnEQL, we consider OWL Lite ontologies, and a subset of SPARQL that includes basic patterns; we do not consider optional patters, union of patterns and filters.

In OnEQL, an ontology is represented in a canonical form which is independent of the specific language used to define it. Accordingly, each ontology is modeled as a deductive database called a Deductive Ontology Base (DOB) composed by an extensional base EOB and an intensional base IOB. Knowledge explicitly described in the ontology is represented in the EOB, while knowledge implicitly encoded is modeled by a set of deductive rules that comprise the IOB. Additionally, to diminish the impact of large sets of explicit and implicit ontology facts in the performance of reasoning and querying tasks, OnEQL provides query optimization and evaluation techniques.

The OnEQL query optimization technique is a twofold strategy that combines cost-based optimization and Magic Sets approaches. The idea of this technique is to first identify an ordering which corresponds to an optimal top-down evaluation of the query, and then apply Magic Sets to transform the DOB into a program specific for the query. The evaluation of the rewritten DOB imitates a top-down computation using a bottom-up strategy [18]. During a bottom-up computation, each fact is computed once. Therefore, if intermediate facts in a query are inferred several times, the bottom-up computation of the rewritten DOB w.r.t. the optimal ordering, can be more efficient than the top-down computation of this ordering.

To identify an optimal ordering of a query, the cost-based optimization technique is defined in terms of a dynamic programming algorithm. To traverse the space of the plans of a query, the algorithm uses a cost model that estimates the cost of evaluating a plan. The cost is defined as an estimate of the number of facts inferred during the execution of the corresponding plan [21].

On the other hand, the Magic Sets approach transforms the DOB into a program where the selections in the query are pushed down into the program, and the number of intermediate facts required to answer the query is minimized.

In this paper we explain the mechanisms implemented in OnEQL, and report their behavior for the ontologies Galen and EHR_RM. The paper is composed of four additional sections. In Section 2, OnEQL is described. Section 3 reports our experimental results for query and reasoning tasks in synthetic and real-world ontologies. In Section 4, we compare existing approaches. Finally, we give our conclusions in Section 5.

2 The OnEQL System

The OnEQL system develops evaluation strategies, and cost-based and heuristic-based optimization techniques for Web ontologies. Its main features are the following:

– Ontologies may be loaded and browsed. The interface is based on the SWOOP Mindswap Project [11].
– Ontologies are translated to the DOB canonical form according to the language in which the ontology is defined. Currently we work the OWL Lite [14] ontology language.
– It offers two modes for queries: SPARQL queries written by the user, or queries expressed in the canonical form. With SPARQL, a user can query RDF triples that encode ontologies written in OWL Lite. The query engine does inference during query evaluation when it encounters the (translated) intensional IOB meta-predicates. To test the OnEQL optimization techniques, the system presents fifty randomly-generated DOB queries which the user may execute. The original and optimized queries are presented, and also their evaluation cost. Additionally, the number of results is indicated and the first fifty results are displayed.

In Figure 1, we present the OnEQL architecture. The techniques implemented in this system have been previously reported in [21].

The OnEQL architecture is comprised of two main components: a query engine and an ontology manager. The query engine evaluates user queries against a specific OWL ontology, and outputs the set of facts that satisfy the query in the input ontology. It is composed of a query parser, a query optimizer and an execution engine. On the other hand, the ontology manager translates OWL ontologies into the OnEQL canonical representation and extracts the statistics that describe the ontologies.

In OnEQL, an OWL ontology is modeled as a deductive database of meta-level predicates called a Deductive Ontology Base (DOB). The extensional database comprises all the ontology statements that represent the explicit ontology knowledge. The intensional database corresponds to the set of deductive rules that define the semantics of the ontology language. Specifically, we represent an OWL Lite ontology as a DOB knowledge base, and a SPARQL query as a DOB query. It should be noted that our current version of OnEQL only considers SPARQL basic patterns.

Table 1 illustrates the EOB and IOB built-in predicates for an OWL Lite subset[1]. Note that some predicates refer to domain concepts (e.g., `isClass, areClasses`), and some to instances (e.g., `isIndividual, areIndividuals`).

There are two catalogs: one stores the DOB, and the other maintains statistics that describe the facts encoded in the DOB. Among the statistics we can mention: cost of inferring implicit facts, cardinality of explicit and implicit facts, and number of different values of each attribute. The Analyzer extracts these statistics from the ontology for explicit and implicit facts, and stores them into the catalog.

A hybrid cost model is used to estimate the cardinality and evaluation cost of the DOB predicates that represent the ontology's explicit and implicit facts [21]. Explicit fact estimates are computed using traditional relational database cost

[1] We assume that the class *owl:Thing* is the default value for the domain and range of a property.

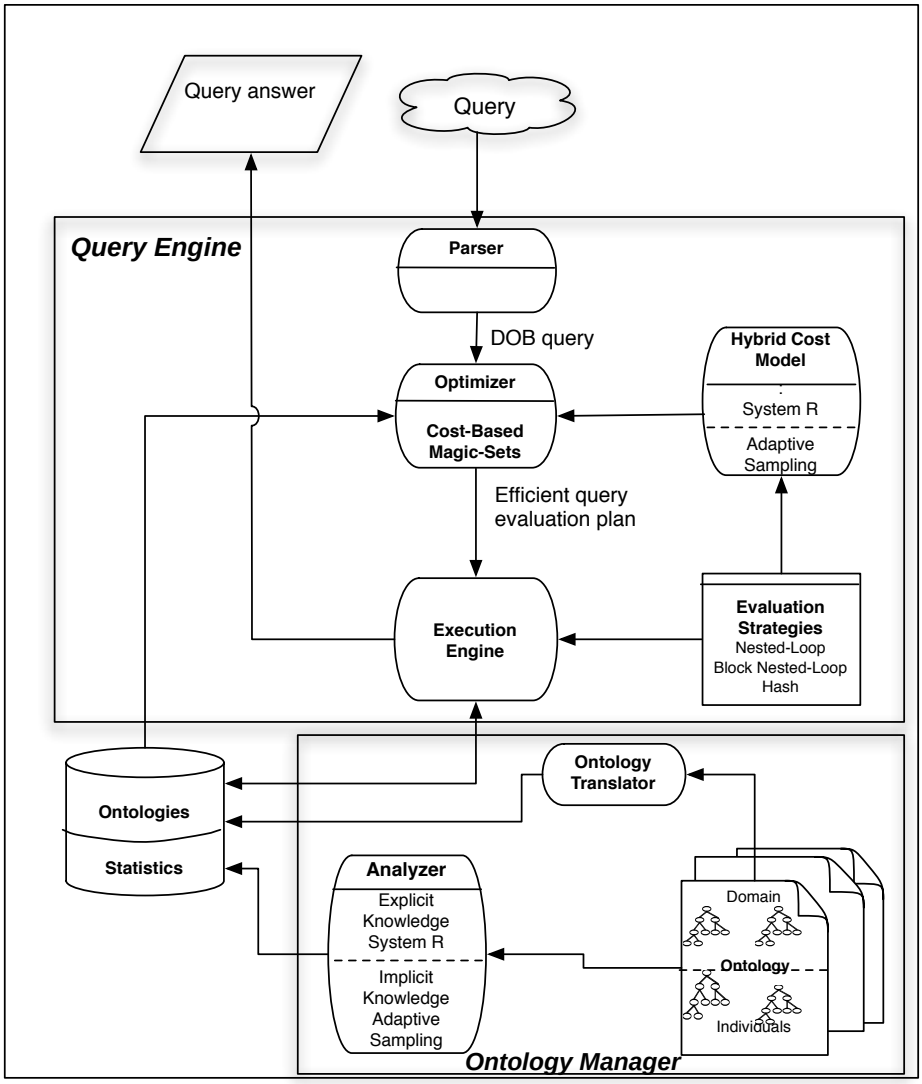


Fig. 1. The OnEQL Architecture

EOB PREDICATE	DESCRIPTION
isOntology(O)	An ontology has an Uri O
isImpOntology(O1,O2)	Ontology O1 imports ontology O2
isClass(C,O)	C is a class in ontology O
isOProperty(P,D,R)	P is an object property with domain D and range R
isDProperty(P,D)	P is a datatype property with domain D
isTransitive(P)	P is a transitive property
subClassOf(C1,C2)	C1 is a subclass of C2
AllValuesFrom(C,P,D)	C has property P with all values in D
isIndividual(I,C)	I is an individual belonging to class C
isStatement(I,P,J)	I is an individual that has property P with value J
IOB PREDICATE	**DESCRIPTION**
areSubClasses(C1,C2)	C1 are the direct and indirect subclasses of C2
areImpOntologies(O1,O2)	O1 import the ontologies O2 directly and indirectly
areClasses(C,O)	C are all the classes of an ontology and its imported ontologies O
areIndividuals(I,C)	I are the individuals of a class and all of its direct and indirect superclasses C; or I are the individuals that participate in a property and belong to its domain or range C, or are values of a property with all values in C

Table 1. Some built-in EOB and IOB Predicates for a subset of OWL Lite

models. Conversely, to estimate the cost and cardinality of data that do not exist a priori, which is the case of the implicit facts, sampling techniques are applied. In our cost model, evaluation cost is measured in terms of the number of intermediate inferred predicates, and the cardinality corresponds to the number of valid instantiations of the predicate. This model estimates the cost and cardinality of explicit and implicit facts, as follows :

- To estimate the cardinality and cost of the intensional predicates that represent implicit facts, we have applied the Adaptive Sampling Technique [13]. This method does not need to extract, store or maintain information about the data that satisfy a particular predicate, and does not make any assumptions about statistical characteristics of the data, such as distribution. Sampling stop conditions are defined to ensure that the estimates are within an appropriate confidence level.
- To estimate the cardinality and cost of the extensional predicates, and the cost of a query plan, we use a cost model *à* la System R [23]. Similarly to System R, we store information about the number of ground facts corresponding to an extensional predicate, and the number of different values (constants) of each predicate variable. Regarding queries, the formulas for computing the cost and cardinality are similar to the different physical *join* formulas in relational queries.

Once a query is received by OnEQL, the parser checks if it is correct. If so, the query is translated into the OnEQL canonical form: the patterns in the *WHERE* clause of a SPARQL query are translated to a conjunctive query, where each pattern corresponds to an EOB or IOB predicate, and the join or conjunction between two predicates represents the '.' (AND) SPARQL operator.

The DOB query is then passed to the optimizer. The optimizer implements a twofold optimization technique and uses the statistics stored in the catalog to identify an efficient query execution plan. Next, the plan is given to the query

engine which evaluates it against the ontology. Facts that satisfy the conditions expressed in the query are returned to the user.

The twofold optimization technique combines cost-based optimization and Magic Sets techniques. In the first stage, the cost-based optimization technique extends the System R dynamic-programming algorithm by identifying orderings of the EOB and IOB predicates in a query. During each iteration of the algorithm, the best intermediate sub-plans are chosen based on the cost and the cardinality that were estimated using our hybrid cost model. In the last iteration of the algorithm, final plans are constructed and the best plan is selected in terms of the estimated cost. This optimal ordering reflects the minimization of the number of intermediate inferred facts using a top-down evaluation strategy. For more details refer to [21].

In the second stage, OnEQL applies Magic Set optimization techniques [19] to the execution plan obtained in the first stage. Magic Sets combines the benefits of both, top-down and bottom-up evaluation strategies and tries to avoid repeated computations of the same subgoals, and unnecessary inferences. The DOB program is rewritten w.r.t. the optimal execution plan, and then evaluated with a bottom-up strategy. "Magic predicates" are inserted into the program to represent bounded arguments in the query, and "Supplementary predicates" are included to represent sideways information-passing in rules. It should be noted that we implemented the general Magic Sets technique for Datalog with the two improvements suggested by [1] to eliminate the first and last redundant supplementary predicates, and to merge consecutive sequences of EOB predicates in rule bodies.

Finally, three different physical operators or evaluation strategies can be used by OnEQL to implement the sideways passing of bindings between two predicates in an execution plan: nested-loop join, block nested-loop join and hash join [18]. The nested-loop join corresponds to a top-down Datalog evaluation strategy where the join variables in the second predicate are instantiated through the sideways passing of information. In the worst case, all of the predicate will be searched; however, more efficient search options may index the predicates by one or more of their arguments. The hash join strategy takes into account the availability of a hash function; it limits the number of pairs of predicate instantiations that need to be compared; nevertheless, it is restricted by the amount of main memory available. These algorithms were developed in the same spirit of relational join operator algorithms; accordingly, relational cost formulas have been modified to reflect the behavior of our operators and to measure the number of intermediate inferred facts; also, implementation details like the use of main memory and pipelining, and the availability of physical structures were represented in these formulas [21]:

- Nested-Loop Join
  For each valid instantiation in the first predicate, we retrieve the matching instantiations in the second predicate, i.e., the join arguments[2] are instantiated in the second predicate through the sideways passing of bindings.

[2] The join arguments are the common variables in the two predicates.

– Block Nested-Loop Join
  The first predicate is evaluated into blocks of fixed size, and then each block is joined with the second predicate.
– Hash Join
  A direct access table is built for the first predicate according to its join argument values. The valid instantiations of both predicates with the same key are joined.

We illustrate the functionality of OnEQL with the following example. In Fig. 2, we present a portion of the Galen ontology expressed in OWL and visualized using the OnEQL interface. A portion of the Galen translated DOB ontology can be seen in Table 2.

DOB predicate
isClass('factkb:Abdomen','Ontologies:galen.owl')
isClass('factkb:AbdominalAorta','Ontologies:galen.owl')
isFunctional('factkb:hasAbnormalityStatus')
isProperty('factkb:actsOn','Ontologies:galen.owl')
isTransitive('factkb:hasCause')
someValuesFrom('factkb:AdhesivePericarditis','factkb:hasOutcome','factkb:Adhesion')
subClassOf('factkb:AdductorMagnus','factkb:NAMEDMuscle')
subClassOf('factkb:AdductorTubercle','factkb:Eminence')
subPropertyOf('factkb:hasLayer','factkb:StructuralPartitiveAttribute')
subPropertyOf('factkb:hasLeftRightSelector','factkb:hasPositionalSelector')

Table 2. Portion of Galen DOB Ontology

Consider the simple query: "Name all the drugs that act on the pathologies caused by the Helicobacter Pylori bacteria". The SPARQL representation of this query is as follows:

```
PREFIX rdfs:<http://www.rdf.org/0.1/>
PREFIX galen:<http://example.org/factkb#>
SELECT ?y
WHERE  {galen:actsOn rdfs:domain ?y.
        galen:actsOn rdfs:range ?x.
        galen:isCauseOf rdfs:domain galen:HelicobacterPylori.
        galen:isCauseOf rdfs:range ?x}
```

This example can be expressed as the following DOB query:
$q(Y) \leftarrow isDomProperty('actsOn', Y), isRanProperty('actsOn', X),$
$isDomProperty('isCauseOf', 'HelycobacterPylori'),$
$isRanProperty('isCauseOf', X).$

The `WHERE` clause is comprised of four triple patterns: the first and second patterns denote the relationship between a drug and a pathology, and the third and the fourth patterns represent the relationship between the pathologies caused by the Helicobacter Pylori bacteria. The result of the query evaluation is a set of solutions, i.e., the matchings of the query patterns and the RDF data.

Considering the triples encoded in Galen, and without taking into account any optimization technique, the evaluation of this simple query will require

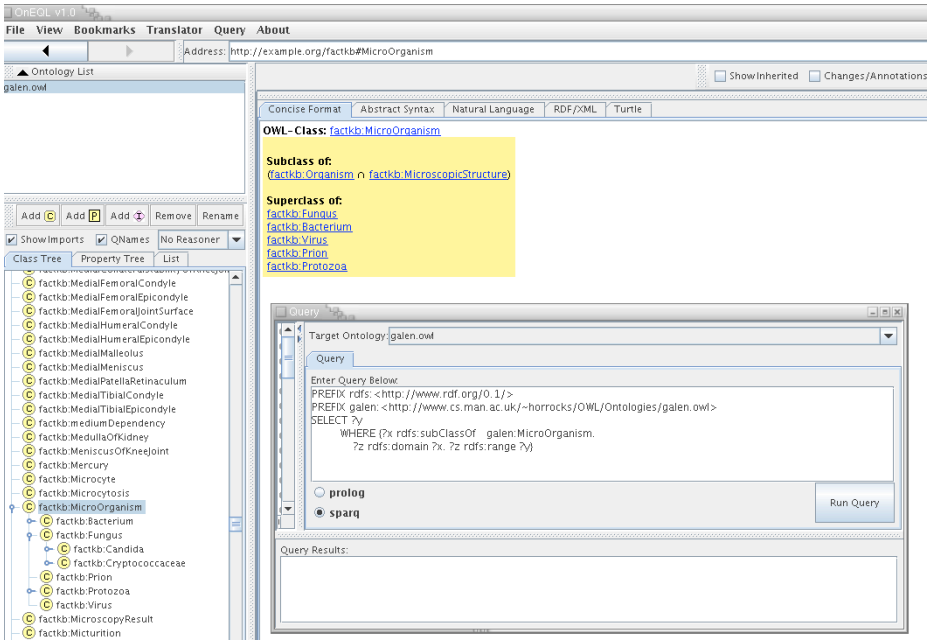


Fig. 2. Galen in OnEQL

727,547 intermediate inferred facts. In this naive plan, first all the combinations of drugs and pathologies are considered, and then the pathologies caused by the Helicobacter Pylori are selected. To reduce the number of intermediate computed facts, a cost-based optimization technique estimates the cost of the different orderings of the evaluation of the query, and recognizes a better way to evaluate the query. Therefore, it produces an execution plan where first, the different pathologies caused by the Helicobacter Pylori are selected; following this, the drugs that act on these pathologies are projected out. On the one hand, there are seventeen instances of the relationship between drugs and pathologies that require 726,980 inferences to be produced; on the other hand, the Helicobacter Pylori is only related to five pathologies and 72,743 inferences are needed to project out these pathologies. Thus, this new execution plan is less expensive in time and in the number of intermediate inferred facts, 291,371.

The optimal DOB query ordering follows:

$q(Y) \leftarrow isDomProperty('isCauseOf','HelycobacterPylori'),$
$isRanProperty('isCauseOf', X),$
$isDomProperty('actsOn', Y), isRanProperty('actsOn', X).$

Once an optimal query ordering has been selected, query bindings can be used to rewrite the canonical representation of the ontology and simulate the pushing of selections that occurs in a top-down evaluation strategy. In this example, the query has nine bindings, e.g., "rdfs:domain" and "galen:actsOn" in

the first pattern. Besides rewriting the program with supplementary and Magic predicates, the query is also rewritten to include the "seed" that represents the variable bindings. The rewritten program and query are then evaluated using a semi-naive bottom-up evaluation strategy. The Magic Sets rewritten DOB of the optimal ordering required 111,071 intermediate inferred facts during the bottom-up evaluation, while the top-down evaluation of this ordering required 291,371 inferred facts.

3 Experimental Results

In this section we report the behavior of the OnEQL query techniques in ontologies commonly used in the health domain. We consider the ontologies Galen [6] and EHR_RM [5].

The Galen ontology is a repository of medical terms and procedures. It provides a set of modeling conventions and patterns that have proved sufficiently robust to be applied in practical developments such as surgical terminologies, drug information and data entry systems. Particularly, it has been used for the development of the French national classification of surgical procedures CCAM [20] and for the development of the drugs ontology in the UK [24].

The EHR_RM ontology is a controlled vocabulary for electronic health records that maintains all the information required to facilitate the flow needed for patient care. EHR_RM is comprised of two levels: level one corresponds to a set of classes and relationships that represent properties in the whole world; level two is composed of a set of clinical concepts which are related to the general concepts in level one.

In Table 3, these ontologies are described in terms of the number of classes, the average properties associated with a class, the maximal fan out, the height of the ontology, and the number of parents. We can observe that Galen is a hierarchy of concepts where each class can have a large number of sub-classes; almost no relationships or properties are associated with each class. EHR_RM is simpler and there are some relationships and properties related to a class. These characteristics impact on the evaluation cost of queries that require recursive traversals of the data.

Ontology	#Classes	Fan out	Height	# Parents
Galen	2749	18	2	13
EHR_RM	187	2	2	7

Table 3. Ontology descriptions

We conducted an experimental study to analyze the behavior of our query techniques on synthetic ontologies and the above-mentioned real-world ontologies. A synthetic ontology document was generated with ten related ontologies and a total of 4350 basic facts. Each ontology has between twenty to thirty

classes, around twenty relationships, three to five attributes for each class, and around sixty sub-class relationships. All the numbers described above were randomly chosen following a uniform distribution. Additionally, we randomly generated sixteen chain queries[3] for each ontology. Experiments were executed on a Sun Fire V440 equipped with two UltraSPARC IIIi processors running at 1.593 GHZ with 16 GB RAM. The OnEQL system was implemented in Java 1.4 and SWI-Prolog 5.6.1. In this paper we report the predictive capability of the cost model, and cost improvements from using the twofold optimization strategy.

First, we report the correlation between the estimated cost of the top-down evaluation of 384[4] orderings, and the actual cost of evaluating the Magic Sets rewritings w.r.t. these orderings using a bottom-up strategy. The idea is to measure if the estimated cost of the top-down ordering is correlated to the actual cost of applying Magic Sets to this ordering, i.e., Magic Sets emulates a top-down evaluation strategy but tries to avoid repeated computations of the same subgoals. The correlation for the real-world ontology Galen is 0.53, while for EHR_RM it is 0.43.

Additionally, we studied the benefits of the twofold optimization strategy. For each query we applied Magic Sets to all its orderings, and we compared:

- The percentile of the Magic Sets optimal ordering actual cost, i.e., the cost of applying Magic Sets to the optimal ordering. For the three ontologies we can observe that the cost of the Magic Sets optimal ordering falls in at least the 74th percentile, indicating that three quarters of all the orderings are worse than this cost (Table 4).
- The average ratio of the cost of the Magic Sets optimal ordering to the worst cost (resp. median cost), i.e., the number of times the worst cost (resp. median cost) contains the optimal cost; it is expressed as a percentage (Figures 3 and 4).

Ontology	Percentile
Synthetic	74th
Galen	77th
EHR_RM	75th

Table 4. Percentile of the cost of the optimal ordering

For the synthetic ontologies, the average of the ratio of the optimal cost with respect to the worst-case is 45%; Galen and EHR_RM have averages of 15% and 32%, respectively. The averages of the ratio of the optimal cost with respect to the median are 74%, 63% and 81% for synthetic, Galen and EHR_RM ontologies respectively. From these results we can conclude that the cost of an optimal ordering is always better than the median cost, while the optimal cost

[3] Queries where bindings are propagated from left to right in a chain-like fashion
[4] Each of the sixteen queries has four sub-goals, $16 \times 4! = 384$

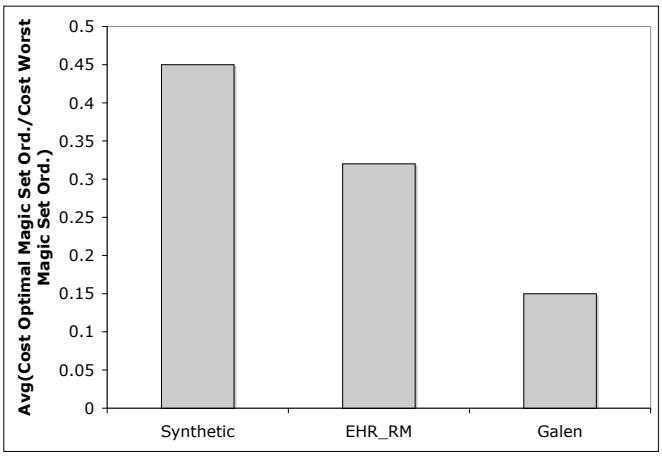


Fig. 3. Average ratio of the cost of the Magic Sets optimal ordering to the worst cost

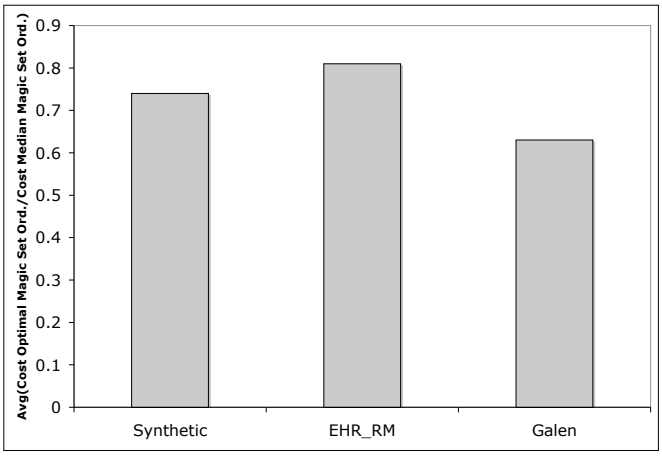


Fig. 4. Average ratio of the cost of the Magic Sets optimal ordering to the median cost

corresponds to a small fraction of the worst cost. To explain these results, recall that classes in Galen are more connected than classes in EHR_RM and, in consequence, the same fact may be generated several times during the computation of the transitive closure of Galen's subsumption relationship. Therefore, our proposed optimization techniques may have a better chance of causing an impact on queries against Galen (by minimizing the number of intermediate and duplicated inferred facts), than on queries against simpler ontologies like EHR_RM.

4 Related Work

Efficient query evaluation techniques against ontologies have been proposed in [3, 7–10, 16, 21, 22]. In [3, 8, 9], relational query techniques and Description Logics reasoning services have been combined to efficiently solve querying and reasoning tasks over individuals of an ontology stored in a database. These systems are built upon relational DBMS and they do not develop optimization techniques that use the semantics encoded in the ontology to identify good evaluation plans.

In [22], ontology segmentation techniques are proposed to approach the problem of querying large ontologies such as Galen. These techniques exploit the semantic connections between ontology terms to enable users to create new sub-ontologies with the portion of the original ontology that is relevant to the application or query. On the other hand, the projects described in [10, 16] developed Magic Sets query rewriting techniques to generate new programs that evaluate the input query more efficiently. In none of these two techniques cost or cardinality estimations are considered, and the new portion of the ontology or the evaluation strategy may be inefficient depending on the shape of the ontology.

5 Conclusions and Future Work

In this paper we have described OnEQL, a tool that evaluates SPARQL queries against OWL Lite ontologies. We implemented these two standards because they achieve good trade-offs between expressiveness and computational tractability.

To enhance the performance of the reasoning and querying tasks, we propose a twofold optimizer which combines the benefits of cost-based and Magic Sets approaches. Additionally, a hybrid cost model is implemented. This cost model integrates estimation techniques used in traditional relational DBMSs [18, 23] with adaptive sampling to estimate the cost or cardinality of explicit and implicit classes [13]; the cost model allows the precise estimation of these metrics.

In our experiments we observed that correlations between estimated and actual values are not greater than 0.53. From these values, we can conclude that our cost model overestimates the cost if the top-down evaluation produces a large number of repeated inferences, because the actual cost of applying Magic Sets emulates a top-down evaluation without repeated inferences.

Also, the implemented optimization techniques allow the identification of optimal query plans whose cost is less than 45% of the cost of the worst plan.

In the future, we plan to conduct experiments on other large ontologies, and to define cost metrics that provide a better estimate of the behavior of the Magic Sets technique.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.
2. D. Brickley and R. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/, 2004.
3. D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tailoring OWL for data intensive ontologies. In *Proc. of the Workshop on OWL: Experiences and Directions*, 2005.
4. Disease Ontology. http://diseaseontology.sourceforge.net.
5. EHRRM Ontology. http://trajano.us.es./ isabel/EHR/EHRRM.owl.
6. GALEN Common Reference Model. http://www.openclinical.org/dld_galenCRM.html.
7. B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proc. of the WWW2003: World Wide Web Conference*, 2003.
8. I. Haarslev and R. Moller. Optimization techniques for retrieving resources described in OWL/RDF documents, First results. In *Proc. of KR2004: International Conference on the Principles of Knowledge Representation and Reasoning*, 2004.
9. I. Horrocks and D. Turi. The OWL Instance Store: System description. In *Proc. of CADE2005: International Conference on Automated Deduction*, 2005.
10. U. Hustadt and B. Motik. Description Logics and Disjunctive Datalog The Story so Far. In *Proc. of DL 2005 - International Workshop on Description Logics*, 2005.
11. A. Kalyanpur and E. Sirin. SWOOP - A Hypermedia-based Featherweight OWL Ontology Editor. http://www.mindswap.org/2004/SWOOP/, 2004.
12. G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *Proc. of the WWW2002: World Wide Web Conference*, 2002.
13. R. Lipton and J. Naughton. Query size estimation by adaptive sampling (extended abstract). In *Proc of SIGMOD1990: Special Interest Group on Management of Data Conference*, 1990.
14. D. McGuinness and F. van Harmelen. OWL Web Ontology language overview. *W3C Recommendation*, 2004.
15. Medical Subject Heading (MeSH). http://www.nlm.nih/gov/mesh.
16. B. Motik, R. Volz, and A. Maedche. Optimizing Query Answering in Description Logics using Disjunctive Deductive Databases. In *Proc. of the KRDB2003: International Workshop on Knowledge Representation meets Databases*, 2003.
17. E. Prudhommeaux and A. Seaborne. SPARQL Query Language for RDF. In *http://www.w3.org/TR/rdf-sparql-query*, 2006.
18. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. Mc Graw Hill, 2003.
19. R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
20. J. Rodrigues, B. Trombert-Paviot, R. Baud, J. Wagner, P. Rusch, and F. Meusnier. Galen-In-Use: an EU Project applied to the development of a new national coding system for surgical procedures: NCAM. In *Medical Informatics Europe*, 1997.

21. E. Ruckhaus, E. Ruiz, and M. Vidal. Query Evaluation and Optimization in the Semantic Web. In *Proc. of ALPSWS2006: International Workshop on Applications of Logic Programming to the Semantic Web and Semantic Web Services*, 2006.
22. J. Seidenberg and A. Rector. Web Ontology Segmentation Analysis, Classification and Use. In *Proc. of WWW2006: World Wide Web Conference*, 2006.
23. P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. *Proc. of SIGMOD1979: Special Interest Group on Management of Data Conference*, 1979.
24. M. Stearns. SNOMED clinical terms: overview of the development process and project status. In *Proc. of AMIA2001: American Medical Informatics Association (AMIA) Symposium*, 2001.
25. XML Schema. http://www.www.w3.org/XMLSchema.

Contextual Logic Programming for Ontology Representation and Querying

Nuno Lopes, Cláudio Fernandes, and Salvador Abreu

Universidade de Évora

Abstract. The system presented in this paper aims at using Contextual Logic Programming as a computational hub for knowledge modeled by web ontologies and enable querying that representation. The components required to behave as a SPARQL query engine are explained and examples of semantic integration of different sources are shown.

1 Introduction

The Semantic Web [6] topic currently represents one of the most active and exciting research areas in computer science. The standard web page provides data oriented for human comprehension, which means a computer agent cannot easily reason about that information. The Semantic Web is a natural evolution of the Internet and, hopefully, will provide the foundations for intelligent systems and agent layers over the World Wide Web.

One important step towards the fulfilling of this vision is the emergence of systems that cannot only understand and reason over Semantic Web documents but also retrieve and process knowledge of multiple information sources. This represents the motivation and purpose of our work which is to use contextual constraint logic programming [2] as a framework for Semantic Web agents, in which knowledge representation and reasoning for ontology documents can be carried out. As such, we adopted the framework Prolog/CX partly described in [3] which makes use of persistence and program structuring through the use of contexts [2]. Throughout this paper, we describe a prototype implementation of a Semantic Web system with three main components:

- A core that is capable of representing web ontologies,
- A SPARQL agent which can answer SPARQL queries about ontologies,
- A back-end capable of mapping Prolog/CX to SPARQL queries, thereby able to query external Semantic Web agents, returning the results as bindings for logic variables present in a Prolog/CX program.

Web Ontology Languages: The Semantic Web is based on existing standard technologies such as XML, RDF and RDF-Schema [14]. Although RDF Schema provides additional modeling primitives, like classes and properties, that enable the hierarchical organization of Web documents, a richer ontology modeling language was necessary. DAML-OIL [8] was then taken as the starting point for

the W3C Web Ontology Working Group in defining OWL [15], the language that is aimed to be the standardized and broadly accepted ontology language for the Semantic Web [4]. OWL is defined as an extension of a sub set of the RDF vocabulary and is divided into three species [9]: OWL Lite, OWL DL and OWL Full.

Query Languages: An open research issue has been the specification of a standard query language that can access this kind of data. There are a variety wide of Semantic Web query languages [11], ranging from pure *selection languages* with limited expressivity to general purpose languages supporting different data representation formats and complex queries. Among all the possibilities, we chose to follow the W3C working groups proposed standard: SPARQL [17], an RDF query language and protocol.

The work presented herein is an extension of what was described in [7], we explain some of the implementation choices and introduce some real world examples. The remainder of this article is structured as follows: Contextual Logic Programming is briefly approached in Section 2 and, in Section 3 we discuss the knowledge representation and ontology querying using Contextual Logic Programming. Section 4 describes a possible approach to implementing a SPARQL agent using the CxLP framework and the issue of querying remote SPARQL agents from within the CxLP framework is discussed in Section 5. Section 6 presents examples of use for the implemented system. Finally, Section 7 provides initial conclusions and possible directions for future research.

2 Contextual Logic Programming

Contextual Logic Programming (CxLP) is a simple yet powerful extension to the Prolog logic programming language which provides a mechanism for modularity. In CxLP a finite set of Horn clauses with a given name is designated by *unit*. Abreu and Diaz [2] provide a revised specification for CxLP, which emphasizes the OOP aspects by means of a stateful model, allowed by the introduction of unit arguments. We now informally focus on some aspects of CxLP, namely parametric units; a more complete specification can be found in [2].

A unit is a parametric module, constituting the program's static definition block. Unit descriptor terms can be instantiated and collected into a list to form a *context*, which can be thought of as a dynamic property of computations. A context specifies the actual *program* (or theory) against which the *current goal* is to be resolved. In short, it specifies the set of predicates which is applicable. These predicates have definitions which depend on the specific units which make up the context. A more extensive description of CxLP may be found in [2, 3].

GNU Prolog/CX introduces a set of language operators called the *context operators* which modulate the context part of a computation.

In a nutshell, when executing a goal G in a context C, a CxLP Engine will traverse C looking for the first unit u that contains a definition for G's predicate.

G is then executed as if it were regular Prolog, in a new context that is the suffix of the C which starts with unit u. Some of the most used operations and operators in GNU Prolog/CX are:[1]

Context extension: U :> G, this operation extends the current context with unit U and then reduces goal G;
Context switch: C :< G, attempts to evaluate goal G in context C, ignoring the current context;
Supercontext: :^ G, evaluates goal G in the context resulting of removing the top unit from the current context;
Current context: :< C, unifies C with the current context;
Calling context: :> C, unifies C with the calling context

3 System architecture

The implemented system is divided in three parts: the core, a front-end (FE) SPARQL agent and a back-end (BE) that maps Prolog/CX to SPARQL queries. The core system is responsible for representing the ontology, the FE enables the resolution of queries expressed in SPARQL and the BE allows the core (and the FE) to query other SPARQL web services. This architecture is represented in Figure 1. By integrating the core, FE, BE and other Logic Programming frameworks namely ISCO [3], the system will be able to access several heterogeneous sources of information: the ontology, other SPARQL agents or web services and relational databases.

The main objective of the core system is to represent web ontologies with CxLP tools. After an ontology is transformed into Prolog/CX units, the capabilities of that representation are that of pure Prolog with modular program structuring. For instance, we can build a front end that acts as a SPARQL web agent which can receive a SPARQL query over a known ontology, process it against the internal representation and respond with the solution. This representation can also be used to map Prolog goals to SPARQL queries and collect the results as logic variable bindings. These approaches are further discussed in Sections 4 and 5.

3.1 Ontology representation

Ontologies are represented using units: there will be one unit that indicates the elements (classes and properties) of the ontologies, another unit for individuals and one for each OWL class and property. This is illustrated in Figure 2.

The individuals and their property values are represented in the unit `individuals`. This unit stores, for each individual, the class it belongs to and, for each of the individual properties, its value.

Each class and property is defined in a unit named after the class or property. Further information about each of this objects, such as hierarchy and restrictions, can be found in its unit.

[1] For a more detailed and formal description, the reader is referred to [2].

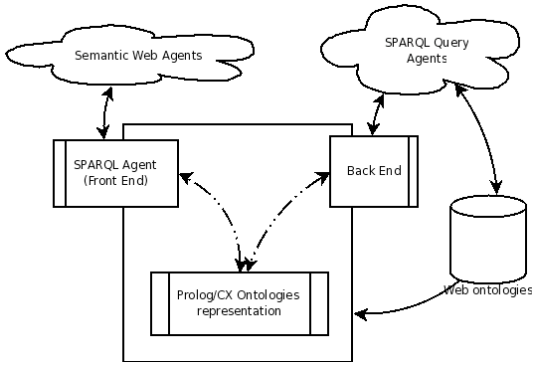


Fig. 1. System architecture

The set of known ontologies are represented in a unit named `ontologies` which lists the classes and properties of each loaded ontology. Each property and class listed in this unit can then be accessed in a uniform manner using the operator `/>`. This operator is defined as a context extension operation, i.e., based on the unit name it constructs a new context in which to evaluate the goal.

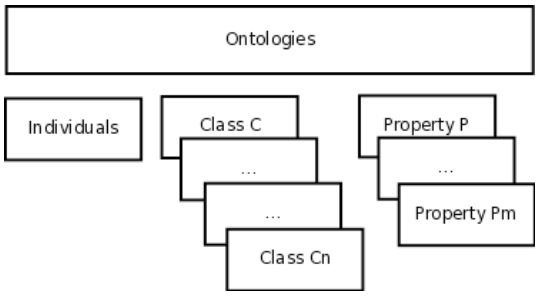


Fig. 2. Ontology representation schema

Ontology Unit This unit represents the ontology information: namespaces, headers, classes and properties. This is done by defining predicates for each case: `ns/3`, `header/3`, `class/2` and `prop/2`. Each predicate contains, in the case of headers and namespaces, an entry with the ontology name, the respective "abbreviation" and its value and, for classes and properties, simply the ontology name and the class or property name.

The information in this unit may be used to query which units belong to the ontology, thereby providing access to all the individuals in the ontology.

Property Units Each property unit contains the information relative to a specific property. The type of the property (datatype or object) and, if specified any other information such as domain and range, property inheritance and property relations.

These properties also define the method to access its value, given the individual name that shall be retrieved previously from the context.

Class Units These units will represent the classes of the ontology and all information relevant to that class. The information includes restrictions on the individual properties and class inheritance.

It also includes a predicate `class_name/1` that provides the name of the current class. This predicate is used in by the query engine to determine the class that the query refers to.

Individuals Unit This unit is the unit that contains all the individuals, its properties and the information about individual relations. The individuals properties are stored as triples, much in the manner of RDF. These properties are defined in the predicate `property/3`. The first argument of this predicate indicates the name of the individual, the second indicates the property and the third argument contains the value of the property for that individual. All the individuals, along with their class, are listed in the predicate `individual_class/2`. Individuals from unnamed classes are not included in this listing: they are only present in the unit that represents the class. This is done to avoid unwanted repetitions when querying the individuals that would be generated if the individuals of the unnamed classes were listed as the other individuals. These individuals are only available in the predicate `individual/1` present in each unnamed class.

There may also be present predicates for defining individual relations, such as `differentFrom/2` and `sameAs/2`, each with individual names as their arguments. These indicate, respectively, that the individuals referred are different or the same [15]. The constructor `owl:AllDifferent` is represented as several `differentFrom` statements, each individual present in the constructor will generate one `differentFrom` statement relating it to every other individual in the list.

3.2 Querying an ontology

The most direct way of retrieving the class individuals is to use the goal `item/1` as shown in Figure 3. There is also a goal `item/0` that has the exact behaviour of `item/1` but has no direct arguments, this predicate, when used with the predicate units in the query will allow to access the property values ignoring the name of the individual.

The `item/1` goal binds, by backtrack, its argument to each individual of the class. There is also the possibility of querying all the individuals in the ontology by omitting a class in the query.

The value of the properties can be accessed by including the unit that represents the property in the context query. This enables selecting only a subset

```
| ?- 'IceWine' /> item(A).
A = 'SelaksIceWine'
```

Fig. 3. Accessing an individual of a class

of the properties. The argument of the property unit will be bound to the value of the property for the corresponding individual, as shown in Figure 4.

```
| ?- 'IceWine' /> hasFlavor(F) :> hasBody(B) :> item(I).
B = 'Medium'
F = 'Moderate'
I = 'SelaksIceWine' ?
```

Fig. 4. Accessing individuals and properties

3.3 Units for refining ontology queries

We propose a number of units which may be used to form queries. We proceed to briefly describe them.

individual/1 Including this unit in the context unifies its argument with the individual name in the same manner as `item/1`. Using this unit provides a more explicit query, by indicating we want the individual name and calling the goal `item/0` instead of `item/1`. Use of this unit is shown in Figure 5.

```
| ?- /> individual(I) :> item.
I = 'WhitehallLanePrimavera' ?
```

Fig. 5. individual example

class/1 If this unit is included in the context it will unify its argument with the class of the matching individual. This is useful to determine the class of the individual when querying the entire ontology, as shown in Figure 6.

property/2 This unit allows to access the properties of the individual without prior knowledge of its name or to query for the property name based on the property value. The first argument is the property name and the second the property value (Figure 7).

```
| ?- /> class(C) :> item(I).

C = 'DessertWine'
I = 'WhitehallLanePrimavera' ?
```

Fig. 6. class example

```
| ?- 'IceWine' /> individual(I) :> property(P,V) :> item.

I = 'SelaksIceWine'
P = locatedIn
V = 'NewZealandRegion' ?
```

Fig. 7. property example

all/2 Including this unit in the execution context is analogous to using a `findall` in Prolog. The first argument is the element and the second will be the list of the elements in the specified form. This allows to retrieve the set of solutions for the variables present in the query, as exemplified in Figure 8.

```
| ?- 'Chardonnay' /> individual(I):> all(I, L) :> item.

L = ['BancroftChardonnay',
     'FormanChardonnay',
     'MountEdenVineyardEdnaValleyChardonnay',
     'MountadamChardonnay',
     'PeterMccoyChardonnay']
```

Fig. 8. all example

optional/1 This unit receives as its argument another unit such as `property/2` or a property unit and will succeed with the results if the unit specified in its argument succeeds. Otherwise it will succeed leaving any variables in its argument unbound. This is similar to the SPARQL `optional` statement [17].

3.4 Native Prolog query representation

To make simple queries easier for Prolog programmes, we created custom predicates that encapsulate the contextual queries. The arguments to these predicates must be defined explicitly after loading the ontology and are follow the conventions:

– Predicate functor is the name of the class
– The first argument is the name of the individual.

The arguments that are present in the predicate after the individual name are specified when defining the predicates. This specification requires indicating the class for which to generate the predicate (that will be the functor of the predicate) and a list of properties that corresponds to the sequence of arguments after the *individual*. This allows the user to choose which properties will be present in the generated predicate.

The generated Prolog representation is listed in Figure 9.

```
'IceWine'(A, B, C) :-
        'IceWine' /> optional(hasMaker(B)) :>
                     optional(hasColor(C)) :>
                     item(A).
```

Fig. 9. generated predicate

This approach is limited because of the fixed arity of the predicates. Some individuals may not have a value for all the properties (an unbound variable for that property will be returned in this case) and other individuals may have properties that are not present in the predicate and thus the user is unable to retrieve its value with these predicates, using this method.

4 A SPARQL agent in CxLP

SPARQL is a Candidate Recommendation for a RDF query language [17]. It is under continued development towards becoming the standard query language for the semantic web [11] and although it is mainly used to query RDF graphs, it can also be used to query an RDF Schema or OWL ontology on the individual and properties level [13, 16, 17].

SPARQL has no inference engine inherent to the language, it merely specifies a syntax for the query and a means for returning the intended information.

The developed system is using SPARQL to query an ontology, allowing access to properties and resulting in individuals and property values.

The implemented SPARQL parser follows the specifications of the language defined in [17] and the results are returned in XML, the format of which is specified in [5]. The parser constructs a Prolog/CX context representing the query; this context is then activated by sending a message to calculate the output and display the resulting XML form. This specification allows our system to be easily made available trough a web service.

SPARQL has 4 types of queries: `select`, `ask`, `construct` and `describe`. The `select` query is used to retrieve the values of the properties and individuals. `Ask`

simply returns a boolean answer depending on the veracity of the query. The `construct` and `describe` are not currently implemented as they would return data as RDF graphs.

The following sections briefly describe the SPARQL query language, the resolution of queries and the XML output of the system.

4.1 SPARQL and mapping examples

The mapping process (SPARQL parser) transforms a SPARQL query into a Prolog/CX context. The execution of this context will bind the variables present in the query with the results.

The context has a similar structure to the SPARQL query, consisting of the following parts, each of which being a parametrized unit:

prefix indicates the default prefix;
from specifies the RDF dataset to query;
select lists the variables that should be present in the output;
where restriction conditions;
Modifiers if present, these modifiers will change the number of results (`limit` and `offset`) and/or their order (`order by`).

The parser receives as input a SPARQL query, shown in Figure 10, and returns the context to be executed (Figure 11).

```
SELECT
       ?flavor ?body
WHERE {
   ?t :hasFlavor    ?flavor  .
   ?t :hasBody      ?body  .
}
```

Fig. 10. Query example (simple select)

4.2 Query resolution system

The query resolution is triggered by evaluating the goal `item` in the context returned by the mapping process. This is akin to sending the message `item` to an object. The core unit in this process is the unit `triple/1` which is responsible for instantiating the variables in the query by accessing the internal representation of the ontology.

The *Modifiers* will alter the query results, their order or number. If no modifiers are present in the query the unit `all` will be included in the context meaning that all the possible bindings will be returned.

```
1 [ all,
2   where([set([
3               triple(A,hasFlavor,B),
4               triple(A,hasBody,C) ])
5         ]),
6  select([flavor=B,body=C]),
7  vars([flavor=B,body=C,t=A]),
8  defs ]
```

Fig. 11. Results of the query example

Also, currently, the `from` clause has no effect since the instantiation is done with an already loaded ontology.

triple/3 The core unit in this process is the unit `triple/3` which is responsible for instantiating the variables in the query by accessing the internal representation of the ontology. The implementation of this unit is shown in Figure 12. It generates one query to the system core for each property that appears in the SPARQL query. The pattern in line 3 of Figure 11 will generate the following query:

```
/> property(hasFlavor,F) :> item(I).
```

The argument of the `item/1` goal will be instantiated with the name of the individual (in this case the variable `I`). The arguments of the unit `property` are the name of the property being queried and the value of that property for the returned individual. Using the `property` unit to query the internal representation has the advantage of being able to perform the query using a variable in the position of the property name (instead of "hasFlavor" in the example).

The results of this query will be bound to the representation of the variables present in the SPARQL query.

```
1 :- unit(triple(S, P, O)).
2
3 item :-
4         /> property(P,O) :> item(S).
```

Fig. 12. Unit triple

Also, currently, the `from` clause has no effect since the instantiation is done with an already loaded ontology.

"A pattern solution can then be defined as follows: to match a basic graph pattern under simple entailment, it is possible to proceed by finding a mapping from blank nodes and variables in the basic graph pattern to terms in the graph being matched; a pattern solution is then a mapping restricted to just the variables, possibly with blank nodes renamed. Moreover, a uniqueness property guarantees the interoperability between SPARQL systems: given a graph and a basic graph pattern, the set of all the pattern solutions is unique up to blank node renaming." in [17].

Each basic operation is represented as a unit.

All the units that are present in the context generated by the SPARQL parser will answer to the goal `item/0` or `item/1` (in case of the unit returning a bound solution). Each unit will then perform the operation it represents based on its arguments and on the result of the parent context.

The units that alter the query results (the *Solution Modifiers*), such as `order by`, `limit` and `distinct` fetch all the bound variables from the parent context collecting them in a list. They them perform its operation on over the elements of the list thus achieving a new list with the results. This will be the final list to be presented as XML.

5 Mapping Prolog to SPARQL Queries

The purpose of having web ontologies represented in a logic contextual form is to create a mechanism to access and work over those ontologies. So far, an ontology mapping engine for local reasoning was introduced. However, for increased flexibility and functionality, one could consider of merging the reasoning of the system internal knowledge base with external ontologies provided by external Semantic Web services. To achieve this goal, a back end was developed that is capable of communicating with SPARQL web agents. This enables writing Prolog/CX programs to reason simultaneously over local and external ontologies.

5.1 Architecture

The back end engine provides additional means to query external Semantic Web services in SPARQL. Although it can be viewed as a single independent component, the objective is to integrate it with the system in a manner that allows a programmer to reason over external and internal ontologies using the same query syntax and declarative context mechanics as the internal system reasoning. This will allow the system using programmer to query transparently internal and external ontologies and merge their results in the same program.

To achieve this level of functionality, a Prolog/CX to SPARQL engine was developed that has the following level of execution capability:

- Translates a Prolog/CX goal into SPARQL;
- Sends the SPARQL query to the indicated Semantic Web SPARQL service;
- Fetch the XML result file, parse it and return the solutions as Prolog variable bindings using the Prolog/CX backtrack mechanism.

It is necessary to provide additional information in order to query the external agent if the SPARQL protocol [18] is to be used. This includes, among others, the `url` of the service, the data format of the response and, possibly, an ontology URI. The latter means that external agents may have capabilities for querying ontologies from any given Internet location, such as the `XML Armyknife` Semantic Web service [10] that is used throughout this section to illustrate the back end functionality. The response format can vary from different types like simple HTML for Internet browsing purposes, or the SPARQL Query Results XML Format [5] for agents like ours.

5.2 Querying an external SPARQL agent

Accessing different Semantic Web external agents rises the problem of implementing a unique interface that can communicate with all of them. The core of this problem has been addressed by the W3C group, which is working on a SPARQL protocol for web agents communication [18]. This W3C *Candidate Recommendation* describes means of conveying SPARQL queries from query clients to a SPARQL query processing service and returning the query results to the requesting entity. Therefore, and after finding some SPARQL agents that rely on this protocol [10, 1], we decided that at this point, for demonstration and proof of concept purposes, the back-end would also rely on the interface communication described on this document.

Communication and query solutions The execution of a back-end query can be described as a three step process. The first is the process of mapping a Prolog/CX query to a SPARQL. The query illustrated in Figure 10 originates the following SPARQL query (Figure 13):

```
1       SELECT ?id ?hasMaker ?hasColor
2       WHERE { ?id :hasMaker ?hasMaker.  ?id :hasColor ?hasColor.}
```

Fig. 13. generated SPARQL code

After the Prolog/CX translation to SPARQL constructs a query, a communication process must be carried out between the back end and the Semantic Web sparql service that is to be used. The back end implements a simple connection model divided into the following steps:

1. Establish connection
2. Send query

3. Receive the response
4. Close connection

After correctly encoding the SPARQL query, the back end will start the communication process with the external agent. This represents the *Establish connection* item in the above list and includes the validation of the the Web service:

- Open the communications via `C` sockets;
- Verify if the external host is up and ready for communication.

After the connection is established, the query is then sent through the socket. If everything went well, the external agent response is then received and flushed into a XML file. Then the connection is closed. This represents the remain items in the back end query execution list.

After receiving the response and closing the connection, the response is saved locally in a XML file and the process returns to the Prolog side, where the file is parsed and processed. The XML format that represents the solutions to the query follows the specification described in the SPARQL Query Results XML Format [5]. This file, which contains all the existing solutions for the query, is then parsed and converted into a Prolog List. Finally, the back end will provide each logic solution to the query present in the response file, one at a time if more than one are available, via the backtracking mechanism.

6 Examples

6.1 Using SPARQL to query a relational database

By using ISCO framework we enable to use SPARQL to query any database. The database used as test in the query example shown in Figure 15 is the database of Universidade de Evora's Information System (SIIUE) [12]. The database contains data relative to the implementation of Academic services. The relation used to query in Figure 15 is represented in Figure 14. It is necessary to define that each field name is prefixed by the name of the relation and an _ forming: `RelationName_FieldName` in order to be able to represent fields with the same name from different relations. This way, the `:student_number` query in line 2 of Figure 15 represents the field `number` of table `student`. The name of the individual, that will be mapped to each tuple in the relation, is the Postgresql internal `OID`[2] of the tuple.

This feature is implemented as an example and there are several improvements to be made in order to make it reasonably efficient. Some are, for instance:

[2] The OID is a unique number across the entire installation automatically assigned to a row and that identifies it. PostgreSQL uses OIDs to link its internal system tables together. Further information can be found in http://www.postgresql.org/docs/8.2/static/datatype-oid.html

```
mutable class student.
  id: individual.id. unique.
  number: int. unique.
  institution: institution.id.
```

Fig. 14. ISCO definition of the relation aluno

```
select * where {
  ?a :student_number ?c .
  ?a :student_institution ?b .
  FILTER ( ?c > 300 && ?c < 500  )
}
```

Fig. 15. using SPARQL to query a relational database

- allow to query more than one relation field. As the query is being translated to the ISCO language it is performing one query to the database for each field present in the SPARQL query. This could be improved by detecting patterns in the query and rewriting it to minimize the number of queries to the database;
- enable filtering the elements before they are retrieved from the relation. In its current stage all the elements are gathered from the relation being queried and filtered later in the `filter` statement.

6.2 SPARQL Web service

As another example of the developed system an example web interface was built in order to allow answering of SPARQL queries over the web.

This has a simple user interface in which users can specify the queries and retrieve the results.

There is also available a version in which the query is specified as part of the URL and the results are then returned to the browser or agent. This form of input is mostly aimed for automatic use by a SPARQL agent.

This form of query method has some shortcomings:

- the core system has to load the ontology for each query it is made (which takes some time);
- the ontology over which the query will be performed is already integrated with the system but, in this case, there is only the possibility of querying the loaded ontology.

The implemented example works under the second assumption. The ontology is integrated with the system and the SPARQL query will be performed over that ontology.

7 Conclusion

The system we described and implemented provides a representation abstraction layer for web ontologies that can be accessed by logic programs.

The selected web languages, SPARQL and OWL, have been shown to be appropriate for the scope of our work: to build a working proof-of-concept system which allows us to experiment with Contextual Logic Programming to represent and query ontologies in a way which draws on Prolog's expressiveness as well as the powerful composition mechanisms of CxLP.

We illustrated how this representation can be used to develop Semantic Web agents by describing two components: a front-end and a CxLP back-end. There are aspects of other, existing systems that will benefit from the ability to query SPARQL sources: for instance, we are working on integrating the SPARQL back-end into the ISCO [3] system.

Future Work Supporting a well-defined OWL sublanguage is necessary in order to provide reliable, trusted semantic web agents which will be usable in wider application sceneries. We are working towards providing provably correct OWL DL compatibility at the reasoning level, over the internal representation. This issue is orthogonal to the rest of the work described herein but it is essential if we expect the system to gain acceptance in the design of SW agents.

The implemented SPARQL agent currently does not cover the full language specification. Although full SPARQL language support is not our immediate intended purpose, we are working towards providing complete support for it.

Another important goal is to provide the core with capabilities to work with several ontologies at a time. Although it is not relevant for the purpose of this work, it is an essential feature for any Semantic Web application software and we purport to use CxLP's versatile modularity mechanisms to effectively deal with this issue. This aspect will be the goal of upcoming work.

References

1. SPARQler. http://sparql.org/sparql.html, 10 October 2006.
2. Salvador Abreu and Daniel Diaz. Objective: in Minimum Context. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, 2003. ISBN 3-540-20642-6.
3. Salvador Abreu and Vítor Nogueira. Using a Logic Programming Language with Persistence and Contexts. In Masanobu Umeda and Armin Wolf, editors, *Declarative Programming for Knowledge Management*, volume 4369 of *LNCS*, Fukuoka, Japan, 2006. Springer.
4. Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.
5. D. Beckett and J. Broekstra. SPARQL Query Results XML Format. W3C recommendation, W3C, April 2006. http://www.w3.org/TR/rdf-sparql-XMLres/.

6. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic Web. *Scientific American*, 284(5), 2001.
7. Nuno Lopes Cláudio Fernandes and Salvador Abreu. On querying ontologies with contextual logic programming. *OWL: Experiences and Directions Third International Workshop*, June 2007.
8. DARPA. http://www.daml.org/. DAML+OIL, 3 February 2007.
9. M. Dean, G. Schreiber, S. Bechhofer, Frank van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference. W3C recommendation, W3C, Feb 2004. http://www.w3.org/TR/owl-ref/.
10. Leigh Dodds. XML Army Knife. http://xmlarmyknife.org/api/rdf/sparql/query, 5 December 2006.
11. Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. Rdf querying: Language constructs and evaluation methods compared. In Pedro Barahona, François Bry, Enrico Franconi, Nicola Henze, and Ulrike Sattler, editors, *Reasoning Web*, volume 4126 of *Lecture Notes in Computer Science*, pages 1–52. Springer, 2006.
12. Joaquim Godinho, Luis Quintano, and Salvador Abreu. Universidade de Évora's Integrated Information System: An Application. In Hans Dijkman, Petra Smulders, Bas Cordewener, and Kurt de Belder, editors, *The 9th International Conference of European University Information Systems*, pages 469–473. Universiteit van Amsterdam, July 2003. ISBN 90-9017079-0.
13. Jena. A Semantic Web Framework for Java. http://jena.sourceforge.net/, 30 November 2006.
14. Frank Manola and Eric Miller. Rdf primer. W3C Recommendation, World Wide Web Consortium, February 2004.
15. Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, World Wide Web Consortium, February 2004.
16. Protégé. Free, open source ontology editor and knowledge-based framework. http://protege.stanford.edu/, 30 November 2006.
17. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006.
18. W3C. SPARQL Protocol For RDF. http://www.w3.org/TR/rdf-sparql-protocol/, 6 October 2006.

Ontology based information integration using Logic Programming

Gergely Lukácsy and Péter Szeredi

Budapest University of Technology and Economics
Department of Computer Science and Information Theory
1117 Budapest, Magyar tudósok körútja 2., Hungary
Phone: +36 1 463-2585 Fax: +36 1 463-3157
{lukacsy,szeredi}@cs.bme.hu
Keywords: descripton logic, CWA, information integration, logic programming

Abstract. We present an information integration system called SINTAGMA which supports the semantic integration of heterogeneous information sources using a meta data driven approach. The main idea of SINTAGMA is to build a so called Model Warehouse, containing several layers of integrated models connected by mappings. At the bottom of this hierarchy there are the models representing the actual information sources. Higher level models represent virtual databases, which can be queried, as the mappings provide a precise description of how to populate these virtual sources using the concrete ones.
This paper focuses on a recent development in SINTAGMA allowing the information expert to use Description Logic based ontologies in the development of high abstraction level conceptual models. Querying these models is performed using the Closed World Assumption as we argue that traditional Open World DL reasoning is less appropriate in the context of database oriented information integration environments.
The implementation of SINTAGMA uses constraints and logic programming, for example, the complex queries are translated into Prolog goals. This allows us to provide functionalities not supported by other integration frameworks.

1 Introduction

This paper presents the Description Logic modelling capabilities of the SINTAGMA Enterprise Information Integration system. SINTAGMA is based on the SILK tool-set, developed within the EU FP5 project SILK (System Integration via Logic & Knowledge) [2]. SILK is a data centered, monolithic information integration system supporting semi-automatic integration on relational and semi-structured sources.

The SINTAGMA system extends the original framework in several directions. As opposed to the monolithic SILK structure, SINTAGMA is built from loosely coupled distributed components. The functionality has become richer as, among others, the system now deals with Web Services as information sources.

The present paper is about a recent extension of the system which allows the integration expert to use Description Logic models in the integration process.

The paper is structured as follows. In Section 2 we give a general introduction to the SINTAGMA system, describing the main components, the SILan modelling language, and the query execution mechanism. In the next section we discuss the description logic extension of SILan: we introduce the syntactic constructs and the modelling methodology. In Section 4 we present the execution mechanism used when querying Description Logic models. In Section 5 we examine related work. Finally, we conclude with a summary of our results.

The examples we use in the upcoming discussions are part of an integration scenario. This scenario represents a world where we attempt to integrate various information sources about writers, painters and their work (i.e. books, paintings, etc.) and present this information in the form of abstract views.

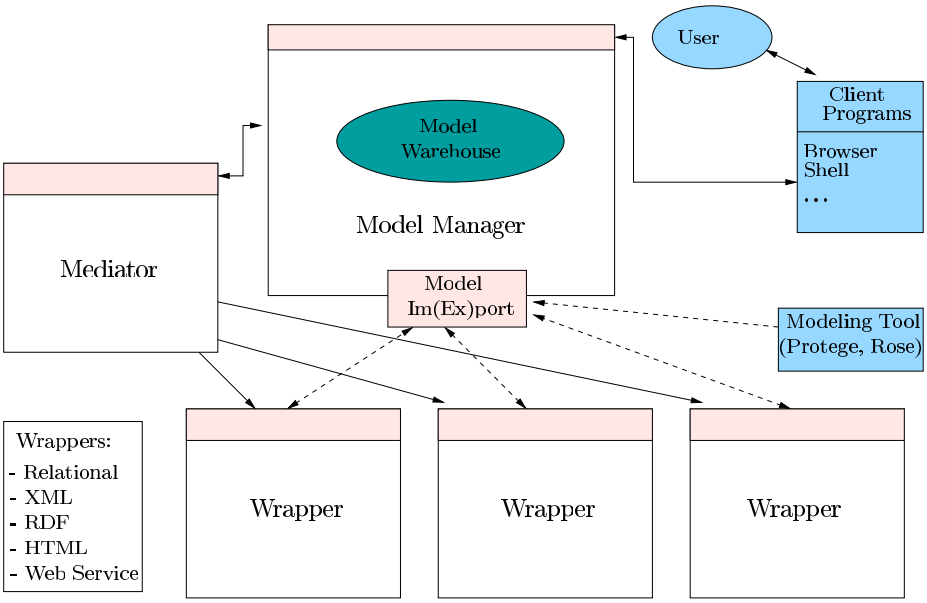


Fig. 1. The architecture of the SINTAGMA system

2 System Architecture

The overall architecture of the system can be seen in Figure 1. The main idea of our approach is to collect and *manage meta-information* on the sources to be integrated. These pieces of information are stored in the *Model Warehouse* of the system, in the form of UML-like models [8], constraints and mappings. This way we can represent structural as well as non-structural information, such as class invariants, implications, etc. The Model Warehouse resides in and is handled by the *Model Manager* component.

We use the term *mediation* to refer to the process of querying SINTAGMA models. Mediation decomposes complex integrated queries to simple queries answerable by individual information sources and, having obtained data from these, composes the results into an integrated form. Mediation is the task of the *Mediator* component.

Access to heterogeneous information sources is supported by *wrappers*. Wrappers hide the syntactic differences between the sources of different kinds, by presenting them to upper layers uniformly, as UML models. These models (called *interface models*) are entered into the Model Warehouse automatically. In the following we give a brief introduction to the main components.

2.1 The Model Manager

The Model Manager is responsible for managing the *Model Warehouse (MW)* and providing integration support, such as model comparison and verification (not covered in this paper). Here we focus on the role of the Model Warehouse.

The content of the MW is given in the language called *SILan* which is based on UML [8] and Description Logics [12]. The syntax of SILAN resembles IDL, the Interface Description Language of CORBA [16]. We demonstrate the knowledge representation facilities of SINTAGMA by a simple SILan example showing the relevant features of the meta-data repository (Figure 2).

```
model Art {
    class Artist:BuiltIns::DLAny {
        attribute String name;
        attribute Integer birthDate;
        constraint self.creation.date > 1900;
    };

    class Work:BuiltIns::DLAny {
        attribute String title;
        attribute String author;
        attribute Integer date;
        attribute String type;
        primary key title;
    };

    association hasWork {
       connection Artist as creator;
       connection Work as creation;
    }; };
```

Fig. 2. SILan representation of the model `Art`

The example describes the model `Art` containing two classes, `Artist` and `Work`. It also contains an association between an artist and her works. We will explain the details of this example within the discussion below.

Semantics of SILan models The central elements of SILan models are classes and associations, since these are the carriers of information. A class denotes a set of entities called the *instances* of the class. Similarly, an n-ary association denotes a set of n-ary tuples of class instances called *links*.

Classes can have *attributes* which are defined as functions mapping the class to a subset of values allowed by the type of the attribute. Classes can inherit from other classes. The instances of the descendant class are all instances of the ancestor class. In our example both `Artist` and `Work` inherit from the built-in class `DLAny` (cf. lines 2 and 8). See Section 3.3 for more details.

Associations have *connections*, an n-ary association has n connections. In an association some of the connections can be named, providing intuitive navigation. For example, the connections of association `hasWork` corresponding to classes `Artist` and `Work` are called `creator` and `creation` respectively (lines 17–18).

Classes are allowed to have a primary key, composed of one or more attributes. This specifies that the given subset of the attributes uniquely identifies an instance of the class. In our example, as a gross simplification, attribute `title` serves as a key in class `Work`, i.e. there cannot be two works (books, for example) with the same title.

Finally, invariants can be specified for classes and associations using the object constraint extension of UML, the OCL language [7]. Invariants give statements about instances of classes (and links of associations) that hold for each of them. The constraint in the declaration of `Artist` (line 5) is an invariant stating that the publication date of each work of an artist is greater than 1900[1]. The identifier `self` refers to an arbitrary instance of the context, in this case the class `Artist`. Then two *navigation* steps follow. In the first step we navigate through the association `hasWork` to an *arbitrary* work of the artist and in the second step from the work to its publication date and state that this is always greater than 1900.

In addition to the object oriented modelling paradigm, the SILan language also supports constructs from the Description Logic (DL) world [12]. This, recently added new feature of SINTAGMA is discussed in Section 3.

Abstractions For mediation, we need mappings between the different sources and the integrated model. These mappings are called *abstractions* because they often provide a more abstract view of the notions present in the lower level models. An example abstraction called `w0` can be seen in Figure 3.

[1] This may be so because the given information source is known to be dealing with works of art of 20th century or later.

This abstraction populates the class `Work` in the model `Art` (line 3) using classes `Product` and `Description`, both from the model `Interface`[2]. This means that the abstraction specifies how to create a "virtual" instance of class `Work`, given that the other two classes are already populated (e.g. they correspond to real information sources). In lines 1–3 the identifiers `m0`, `m1` and `m2` are declared, and these will be used throughout the abstraction specification to denote instances of the appropriate classes.

```
1  abstraction w0 (m0: Interface::Product,
2                  m1: Interface::Description
3              -> m2: Art::Work) {
4
5    constraint
6      m1.id = m0.id and
7      m1.code = 84
8    implies
9      m2.title = m0.title and
10     m2.author = m0.author and
11     m2.date = m0.publication_date and
12     m2.type = m1.description and
13     m2.DL_ID = m0.title;
14 };
```

Fig. 3. SILan representation of the abstraction populating class `Work`

The abstraction describes that given an instance of class `Product` called `m0` and an instance of class `Description` called `m1`, for which the conditions in lines 6–7 hold, there exists an instance `m2` of class `Work` with attribute values specified by lines 9–13[3]. Note that line 6 specifies that the `id` attributes of the two instances have to be the same, and thus corresponds to a relational *join* operation. In our integration scenario `Product` and `Description` actually correspond to real-world Oracle tables containing books and paintings. The task of abstraction `w0` is to convert the records of these tables into instances of class `Work`.

We note that other abstractions can also populate class `Work`. In this case the set of instances of `Work` will be the union of the instances produced by the appropriate abstractions. Note that if a new information source is added, we only have to specify a new abstraction corresponding to this source, while the existing abstractions can be left unchanged.

Notice that the abstraction in Figure 3 takes the form of an implication describing how the given sources can contribute to populating the high level class `Art::Work`. This is characteristic of the Local as View integration approach [4].

[2] In SILan double colons (::) separate the model name from the name of its constituent (class, association, etc.).

[3] Attribute `DL_ID` has a special role as explained in Section 3.3.

2.2 The Wrappers

Wrappers provide a common interface for accessing various information source types, such as relational and object-oriented databases, semi-structured sources (e.g. XML or RDF), as well as Web-services.

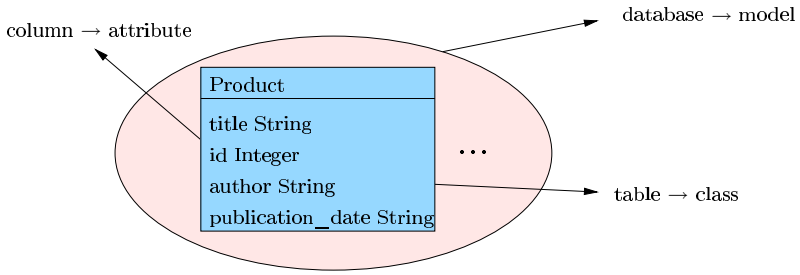


Fig. 4. Modelling relational sources in SILan

A wrapper has two main tasks. First, it extracts meta-data from the information source and delivers these to the Model Manager in the form of SILan models. For example, in case of relational sources, databases correspond to models, tables to classes, columns to attributes, as shown in Figure 4 (cf. class `Product` in abstraction `w0` presented in Figure 3).

The other principal task of a wrapper is to transform queries, formulated in terms of this interface model, into the format required by the underlying information source, and thus allow for running queries on the sources.

2.3 The Mediator

The Mediator [1] supports queries on high level model elements by decomposing them into interface model specific questions. This is performed by creating a query plan satisfying the data flow requirements of the sources. During the execution of this query plan the data transformations described in the abstractions are carried out.

Whenever we query a model element in SINTAGMA, the Model Manager basically provides the following two kinds of information to the Mediator:

1. the query goal itself, i.e. a Prolog term representing what to query;
2. set of mediator rules, using which the mediator can decompose the complex query into primitive ones (i.e. queries that refer only to interface models).

For example, let us consider the query shown below involving class `Work`.

```
query RecentWork
  select *
  from w: Art::Work
  where w.date > 2000;
```

This query is looking for recent works, namely those instances of the class `Art::Work` that were created after 2000[4]. In this case, the query goal is similar to the following simple Prolog expression:

```
:- 'Work:class:220'(DT, [A, B, C, D, E], DA), C > 2000.          (1)
```

Here, the first Prolog goal corresponds to an instance of `Art::Work`. The variables in this term will be instantiated during query execution. The predicate name `'Work:class:220'` is composed from three parts: the kind of the model element (`class`) and its unique internal identifier (220), preceded by the unqualified—and thus non-unique—SILan name (`Work`), provided for readability. Model elements are often referred to by *handles* of form *Kind(Id)*, e.g. `class(220)`. The above predicate name in fact represents the *static type* of the instances queried for.

The first argument of the goal is the dynamic type of the instance, i.e. the handle of the class which, in case of inheritance, can be different from the static type. The second argument contains the values of the static attributes, in this case we have five such variables (cf. declaration of class `Work` in Figure 2), e.g. `C` denotes the value of the attribute `date`. The third and last argument of the query term carries the values of the dynamic attributes. These represent the additional attributes (not known at query time) of the instance if it happens to belong to a *descendant* class of `Art::Work`. Note that in the current implementation of SINTAGMA, as a simplification, the third query argument contains the list of both static and dynamic attributes.

The second part of the query goal corresponds to a simple arithmetic OCL constraint, which uses variable `C` representing the `date` attribute of the work in question.

The mediator rules representing the abstraction `w0` shown in Figure 3 take the following form:

```
'Product:class:190'(_,[A,B,C,D],_),
'Description:class:191'(_,[84,E,B],_) --->
            'Work:class:220'(class(220),[D,A,C,D,E],[D,A,C,D,E])
```

The specific rule above describes how to create an instance of the class `Work` whenever we have two appropriate instances of classes `Product` and `Description` available. If there were more abstractions, the Mediator would get more rules as there would be more than one possible way to populate the given class.

Note that the mediator rules are also used to describe inheritance between model elements. In such a case the dynamic type of the model element on the right hand side of the rule is a variable (as opposed to the constant `class(220)` above). This variable is the same as the dynamic type of the model element on the left hand side. The dynamic attributes are propagated similarly.

[4] We could have created a class named `RecentWork` and populated it by an appropriate abstraction. Then, instead of formulating a SILan query, we could have simply directly asked for the instances of this class. The question whether to use a query or an abstraction is a modelling decision.

Finally, let us mention that an n-ary association is implemented as an n-ary relation, each argument of which is a ternary structure corresponding to a class instance, similar to the one appearing in (1). For example, a query goal for the association `hasWork` (cf. Figure 2) has the following form:

```
:- 'hasWork:association:227'(
       'Artist:class:218'(DT1,[A,B,C],DA1),                    (2)
       'Work:class:220'(DT2,[D,E,F,G,H],DA2)
   ).
```

3 DL modelling in SINTAGMA

Let us now introduce the new DL modelling capabilities of the SINTAGMA system. First we discuss why we need Description Logic models during the integration process and we provide an introductory example. Then we present the DL constructs supported by our system and discuss the restrictions we place on their usage. Finally, we summarise the tasks of the integration expert when using DL elements during integration.

3.1 Introduction

In the Model Warehouse we handle models of different kinds. We distinguish between *application* and *conceptual models*. The application models represent existing or virtual information sources and because of this they are fairly elaborate and precise. Conceptual models, however, represent mental models of user groups, therefore they are vaguer than the application models.

We argue that to construct such models it is more appropriate to use some kind of ontological formalism instead of the relatively rigid object oriented paradigm. Accordingly, we extended our modelling language to incorporate several description logic constructs, in addition to the UML-like ones described earlier. In the envisioned scenario, the high-level models of the users are formulated in description logic and via appropriate definitions they are connected to lower-level models. Mediation for a conceptual model works in the same way as for any other model: the query is decomposed, following the abstractions, until we reach the interface models (in general, through some further intermediate models) which can be queried directly.

Before going into the details, we show an example to illustrate the way how DL descriptions are represented in SILan (note that `Writer` and `Painter` are both descendants of class `Artist`, but otherwise they are normal UML classes.

```
model Conceptual {
  class WriterAndPainter {};
  constraint equivalent {                                      (3)
        WriterAndPainter,
        Unified::Writer and Unified::Painter};
};
```

Here we define the class `WriterAndPainter` by providing a SILan constraint. This constraint can be placed anywhere in the Model Warehouse: in the example above we simply put it in the same model that declares the class `WriterAnd Painter` itself. The constraint actually corresponds to a DL *concept definition axiom*: WriterAndPainter $\equiv$ Writer$\sqcap$Painter. Namely, it states that the instances of class `WriterAndPainter` are those (and only those) who belong to the unnamed class containing the individuals who are both writers and painters. Thus, DL concepts are defined using the Global as View approach [4], as opposed to when populating high-level classes using abstractions (cf. Section 2.1).

Note that the class `WriterAndPainter` could be created without DL support. However, in that case the integration expert would have to go through a much more elaborate process of (1) creating the high level class `WriterAndPainter`, specifying all its attributes and (2) populating it with an appropriate abstraction involving a join. Now, with DL support, she simply formulates a very short and intuitive DL axiom. We argue that this is easier for the expert to do, and it also makes the content of the Model Warehouse more readable to others.

3.2 DL elements in SILan

From the DL point of view, SINTAGMA supports acyclic Description Logic TBoxes containing concept definition axioms formulated in the extension of the $\mathcal{ALCN}(\mathbf{D})$ language (see more below about the extension). Only single atomic concepts, so called *named symbols* can appear on the left hand side of the axioms, such as `WriterAndPainter` in example (3). The remaining atomic concepts, not appearing on the left hand side are called *base symbols*. Such a TBox is definitorial, i.e. the meaning of the base symbols unambiguously defines the meaning of the named symbols. The base symbols, in our case, correspond to normal SINTAGMA classes and associations, e.g. `Writer` and `Painter` in the example (3). The ABox is a set of concept and role assertions, as determined by the instances of the classes which correspond to the base symbols participating in the TBox.

The DL concept constructors supported by SINTAGMA and their SILan equivalents are summarised in Table 1. Note that this table actually describes the possible concept formats on the right hand side of a definition axiom, assuming that we have *expanded* the TBox[5].

The only non-classical DL element in Table 1 is the concrete domain restriction (the last line in the table). Such a restriction specifies a subset of instances of the base concept A for which the given OCL constraint holds. This is a generalisation of the idea of concrete domains in the Description Logics world. Below we show an example of a concrete SILan restriction describing those works whose type (i.e. the value of the attribute `type`) is "`painting`".

```
{class constraint Art::Work satisfies self.type="painting"}
```

The reason we allow only concept *definition* axioms is that we aim to use DL concepts to describe executable high-level views of information sources. In this

[5] The expanded version of an acyclic TBox is another TBox where every named symbol on the right hand side of the axioms is substituted by its definition.

Name	**Syntax**	**SILan equivalent**
Base concept	A	UML class
Atomic role	R	UML association
Top	$\top$	`DLAny`
Bottom	$\bot$	`DLEmpty`
Negation	$\neg C$	`not C`
Intersection	$C \sqcap D$	`C and D`
Union	$C \sqcup D$	`C or D`
Value restrictions	$\forall R.C$	`slot constraint R all values C`
Existential restrictions	$\exists R.C$	`slot constraint R some value C`
Number restrictions	$\bowtie nR$	`slot constraint R cardinality i..j`
Concrete restriction	—	`class constraint A satisfies OCL`

Table 1. (extended) DL concept constructors supported in SILan

sense a DL concept is actually a syntactic variant of a SILan query or a SILan class populated by an abstraction.

Note that this also implies that we use the Closed World Assumption (CWA) in DL query execution. We argue that this is appropriate because of the following three reasons. First, CWA automatically ensures that our DL constructs are semantically compatible with other constructs in the SINTAGMA system. Second, we argue that the Open World Assumption(OWA) is applicable when we have only partial knowledge and would like to determine the consequences of this knowledge, true in every universe in which the axioms of this partial knowledge hold. In contrast with this, in the context of information integration, our users would like to consider a single universe, in which a base concept or a role denotes *exactly* those individuals (or pairs of individuals) which are present in the corresponding database. To illustrate this issue, let us consider the following example: the concept of novice painter is defined to contain painters having at most 5 paintings (for example, being a novice painter may be a precondition for a government grant). To model this situation, the integration expert creates the DL axiom shown below.

$$\mathsf{NovicePainter} \equiv \mathsf{Painter} \sqcap (\leqslant 5\, \mathsf{hasPainting})$$

However, querying this concept, using OWA, will provide no results in general as an open world reasoner would return an individual only if it is *provable* that it has no more than 5 paintings. Practically, this is not what the information expert wants.

The third reason why we decided to use the closed world assumption is the fact that we have huge amounts of data in the underlying databases. Traditional, tableau based DL reasoners do not cope well with large ABoxes [10]. Resolution based DL proving techniques [13] do much better, but they are either still not fast or not expressive enough [15]. By using CWA we can implement DL queries using the well researched, efficient database technology.

3.3 Modeling methodology and tasks of the integration expert

The integration expert is responsible for creating the DL axioms. Although these are represented in SILan within the SINTAGMA system, the expert can use any available OWL editor to create OWL descriptions. These descriptions then can be loaded by the OWL importer of the SINTAGMA system that basically realises an OWL-SILan translation (cf. the "Model Im(Ex)port" box in Figure 1).

One thing the expert should take care of is to match the names of the base symbols and the corresponding SINTAGMA classes and associations. This is often done in two steps: first the integration expert creates concept definition axioms using the widely accepted terminology of the domain, not paying attention to the names of the model elements in the Model Warehouse. Next, the expert provides additional definition axioms for each base symbol connecting it with the proper model element. For example, we could use names `A` and `B` instead of `Writer` and `Painter` in (3), provided that we have the following axioms:

$$\texttt{A} \equiv \texttt{Writer}$$

$$\texttt{B} \equiv \texttt{Painter}$$

Another important issue is to decide how to identify the instances of the base concepts, e.g the instances of the class `Writer` and class `Painter`. Without this, it is not possible to determine the instances of class `WriterAndPainter`.

In a traditional DL ABox, an instance has a name that unambiguously identifies it. Unambiguity is guaranteed because DL reasoning systems use the so called unique name assumption, i.e. they assume that two different instance names denote different elements in the domain.

In SINTAGMA, similarly to databases, an instance is identified by the subset of its attribute values, e.g. two writers could be considered to be the same if their names match. In other words, this means that `name` is a key in class `Writer`.

The problem is that such keys are fairly useless when we compare instances of different sources. This is because, in general, we cannot draw any direct conclusion from the relation of the keys belonging to instances from different classes. For example, databases containing employees often use numeric IDs as keys. Having two employees from different companies with the same ID does not mean that we are talking about the same person. Similarly, if the IDs of the employees do not match, they are not necessarily different persons.

What we need is some kind of shared key that uniquely identifies the instances of the classes participating in DL concept definitions. Luckily, the object-oriented paradigm we use in SINTAGMA provides a nice way to have such identifiers.

We have mentioned earlier that in SINTAGMA the notion of DL concept is a syntactic variant of SINTAGMA class. This also means that the result of

a DL query is an ordinary instance that has to belong to some class(es). For example, when we are looking for the instances that are in both classes `Writer` and `Painter` we are actually interested in an *artist* instance belonging to these classes simultaneously. This is true in general: whatever DL concept constructs we use to describe a DL concept the result must belong to some class that is a common ancestor of the classes involved.

Instead of asking the integration expert to define such common ancestor classes in an ad hoc way, we introduce the built-in class `DLAny`. This class corresponds to the DL concept top ($\top$) and it has only one attribute called `DL_ID` which is a key. We require that all the classes participating in DL concept definitions are the descendants of `DLAny`[6] (cf. lines 2 and 8 of Figure 2). Because of the properties of generalisation attribute `DL_ID` will be a key in all of the descendant classes, i.e. it will exactly serve as the global identifier we were looking for.

Now, the task of the integration expert is to assign appropriate values to the `DL_ID` attributes: she needs to extend the existing abstractions populating the base symbols (classes) to also consider the attribute `DL_ID`. By appropriate values we mean that the `DL_ID`s of two instances should match if these instances are the same, and should differ otherwise. An example for this can be seen in Figure 5 populating the class `Writer`.

```
1  abstraction ap (m0: Interface::Member ->
2                  m1: Unified::Writer) {
3
4     constraint let n = m0.fname.concat(" ").concat(m0.lname) in
5       m1.name = n and
6       m1.birthDate = m0.date and
7       m1.id = m0.iwa and
8       m1.style = m0.style and
9       m1.DL_ID = n;
10 };
```

Fig. 5. Populating the `DL_ID` attribute of a base concept

This abstraction populates the class `Writer` from an interface class called `Member` (lines 1–2). Let us assume that the members of this association have some kind of a unique identifier, such as the membership number of an imaginary "International Writer Association" (IWA), present in the underlying database. It may be worth bringing this key to the class `Writer` (line 7) as it makes possible to find writers efficiently if they happen to be IWA members. However, the unique identifier from the DL point of view has to be different: in fact it is the concatenation of the first and last name of the writer (line 4 and 9).

This is because the class `Writer` can also be populated from other sources where the IWA number makes no sense. Furthermore, we may want class `Writer` to be a descendant of class `Artist`, together with some other classes, such as

[6] Note that this is a necessary condition. As for any concept C, $C \sqsubseteq \top$ holds, any DL instance has to belong to the class corresponding to $\top$, i.e. to `DLAny`.

`Painter`. This requires a key that can be computed from all the underlying sources, such as the name of the artist[7].

To summarise, the integration expert has to perform the following tasks when DL modelling is used during the integration process:

1. declare DL classes and for each provide corresponding definition axioms;
2. ensure that each base concept appearing in the definition axioms is:
   (a) inherited from class `DLAny`,
   (b) populated properly, i.e. its `DL_ID` attribute is filled appropriately.

4 Querying DL models in SINTAGMA

Now we turn our attention to querying DL concepts in SINTAGMA. As described in Section 2.3 our task is to create a *query goal* and a set of *mediator rules*. When we query a DL class, we only generate mediator rules for the base symbols. As these are ordinary classes and associations, this process is exactly the same as the one we use for cases without any DL construct involved.

Recall that a SINTAGMA instance is characterised by three properties, as exemplified by (1) on page 7: its dynamic type `DT`, its static attributes `SA` and its dynamic attributes `DA`. A DL class has only a single static attribute, the `DL_ID` key. However, in contrast with an object oriented query, a DL query may return an answer that has multiple dynamic types. For example, when we enumerate the class `WriterAndPainter` we get instances that belong to both classes `Writer` and `Painter`. Accordingly, an answer to a DL query takes the form of a pair `(ID, DTAs)`, where `ID` is the value of the `DL_ID`, while `DTAs` = `[`$\mathtt{DT}_1$`-`$\mathtt{DA}_1$`,`$\mathtt{DT}_2$`-`$\mathtt{DA}_2$`,...]` is the list of the dynamic types of the answer, each paired with the corresponding dynamic attribute list.

The algorithm specifying what goals to create from a DL concept description is summarised in Figure 6. Here we describe a function Φ_C which, given an arbitrary concept C, returns the corresponding query goal with two arguments, `ID` and `DTAs`. We define this function case by case.

If we have a base class, we simply create a query term representing the instances of the class, similar to the one in goal (1). If we have the intersection of two concepts C and D, we recursively transform concepts C and D and put them in a Prolog conjunction. The union of classes is similar: we create a Prolog disjunction. Negation $\neg C$ is implemented by enumerating the `DLAny` class, and removing those instances which belong to C. The expensive `DLAny` enumeration can be avoided when the negation appears in a conjunction (which is normally the case).

The more interesting cases involve associations. Here R denotes the association itself, while R^D and R^R denote the classes that are the domain and the range of association R, respectively. Recall that a binary association is represented by a binary relation with ternary structures as arguments as in (2).

[7] This is also a simplification. More realistically, the key could be the name together with the birth date.

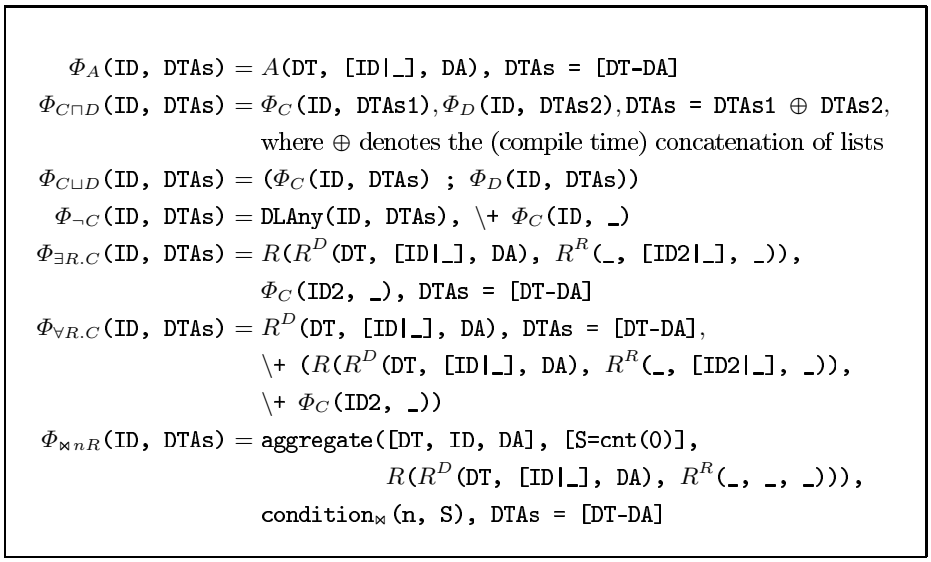


Fig. 6. Transforming DL constructs into query goals

The existential restriction $\exists R.C$ is simply transformed to a query of the association R and the concept C. The goal corresponding to a value restriction $\forall R.C$ first enumerates the domain of R and then uses double negation to ensure that the given instance has no R-values which do not belong to C. Finally, a number restriction $(\bowtie\ nR)$ is transformed into a goal which uses a `bagof`-like Prolog predicate `aggregate/3` to enumerate the instances in the domain of R together with the number of R-values connected to them, and then simply applies the appropriate arithmetic comparison.

A concrete restriction involving a base concept A and an OCL constraint O is transformed in a straightforward way into the query goal as shown below[8]:

Φ_A(ID, DTAs), DTAs = [DT-DA], Ψ_O(ID, DA)

In all formulas so far, `ID` denotes the value of the attribute `DL_ID` containing the unique name of the DL instances (see Section 3.3). Here we make use of the fact that these attributes are always placed first in the static attribute list of an instance. To illustrate the general algorithm, two example transformations are shown in Figure 7. The second example involves an association `hasPainting` and a class `Modern` (representing, say, contemporary pieces of art).

[8] Here, Ψ_O(ID, DA) is the Prolog translation of the OCL constraint O. This is an "old" feature, implemented before the introduction of DL extension into SINTAGMA.

```
Class to query: WriterAndPainter
DL definition:  Writer ⊓ Painter
Query goal:     'Writer:class:234'(DT1,[ID|_],DA1),
                'Painter:class:236'(DT2,[ID|_],DA2),
                DTAs = [DT1-DA1,DT2-DA2]

Class to query: ModernPainterWriter
DL definition:  Writer ⊓ ∃hasPainting.Modern
Query goal:     'Writer:class:234'(DT1,[ID|_],DA1),
                'hasPainting:association:142'(
                      'Artist:class:218'(DT2,[ID|_],DA2),
                      'Work:class:220(_,[ID2|_],_)),
                'Modern:class:237'(_,[ID2|_],_),
                DTAs = [DT1-DA1,DT2-DA2]
```

Fig. 7. Transformation examples

5 Related work

The two main approaches in information integration are the Local as View (LAV) and the Global as View (GAV) [4]. In the former, sources are defined in terms of the global schema, while in the latter, the global schema is defined in terms of the sources (similarly to the classical views in database systems). Information Manifold [14] is a good example for a LAV system. Examples for the GAV approach include the Stanford-IBM integration system TSIMMIS [6].

In SINTAGMA we apply a hybrid approach, i.e. we use both LAV and GAV. When using abstractions to populate high-level classes we employ the LAV, while in case of DL class definitions we use the GAV approach.

There are several completed and ongoing research projects in the area of using description logic-based approaches for both Enterprise Application Integration (EAI) and Enterprise Information Integration (EII) as well.

The generic EAI research stresses the importance of the Service Oriented Architecture, and the provision of new capabilities within the framework of Semantic Web Services. Examples for such research projects include DIP [11] and INFRAWEBS [9]. These projects aim at the semantic integration of Web Services, in most cases using Description Logic based ontologies and Semantic Web technologies. Here, however, DL is used mostly for service discovery and design-time workflow validation, but not during query execution.

On the other hand, several logic-based EII tools use Description Logics and take a similar approach as we did in SINTAGMA. That is, they create a DL model as a view over the information sources to be integrated. The basic framework of this solution is described e.g. in [5,3]. The fundamental difference compared to our approach is that these applications deal with the classical Open World Assumption, therefore their task can be viewed as an ABox instance retrieval task. However, as already discussed in Section 3.2, one problem with this

is that the ABox is distributed among the underlying heterogenous databases which therefore can be extremely big. We argue that existing DL reasoners are not usable when this amount of data and complex DL queries are involved.

6 Conclusions

In this paper we presented the DL extension of the information integration system SINTAGMA. This extension allows the information expert to use Description Logic based ontologies in the development of high abstraction level conceptual models. Querying these models is performed using the Closed World Assumption over the underlying information sources.

We have presented the main components of the SINTAGMA system: the Model Manager which is responsible for the Model Warehouse, the Wrapper, which provides a uniform view over the heterogenous information sources and the Mediator, which decomposes complex high-level queries into primitive ones answerable by the individual information sources.

Next, we have described the DL modelling elements the integration expert can use when building conceptual models and we have also discussed the modelling methodology she has to follow. We have presented the way how DL queries are executed within the SINTAGMA system. Finally, we have illustrated our approach by providing a use case about artists.

We argue that our solution for combining DL and UML modelling in a unified integration framework provides a viable alternative to existing systems. The usage of DL constructs in building high-level conceptual models has substantial benefits, both in terms of modelling efficiency and maintenance.

Acknowledgements

The authors acknowledge the support of the Hungarian NKFP programme for the SINTAGMA project under grant no. 2/052/2004. We would also like to thank all the people participating in this project, and especially Tamás Benkő.

References

1. Liviu Badea and Doina Tilivea. Query Planning for Intelligent Information Integration using Constraint Handling Rules, 2001. IJCAI-2001 Workshop on Modeling and Solving Problems with Constraints.
2. T. Benkő, P. Krauth, and P. Szeredi. A logic based system for application integration. In *Proceedings of the 18th International Conference on Logic Programming, ICLP 2002*. Springer, LNCS, 2002.
3. A. Borgida, M. Lenzerini, and R. Rosati. Description logics for databases. In *Description Logic Handbook*, pages 462–484, 2003.
4. D. Calvanese, D. Lembo, and M. Lenerini. Survey on methods for query rewriting and query answering using views. Tech. report, University of Rome, April 2001.
5. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Daniele Nardi, and Riccardo Rosati. Description logic framework for information integration. In *Principles of Knowledge Representation and Reasoning*, pages 2–13, 1998.

6. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, 1994.
7. T. Clark and J. Warmer, editors. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer, 2002.
8. Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.
9. Vladislava Grigorova. Semantic description of web services and possibilities of BPEL4WS. *Information Theories and Applications*, 13(2):183–187, 2006.
10. V. Haarslev and R. Möller. Optimization techniques for retrieving resources described in owl/rdf documents: First results. In *Ninth International Conference on the Principles of Knowledge Representation and Reasoning, KR 2004, Whistler, BC, Canada, June 2-5*, pages 163–173, 2004.
11. M. Hepp, F. Leymann, J. Domingue, A. Wahler, and D. Fensel. Semantic business process management: A vision towards using semantic web services for business process management, 2005.
12. Ian Horrocks. Reasoning with expressive description logics: Theory and practice. In *Proc. of the 18th Int. Conf. on Automated Deduction (CADE 2002)*, number 2392 in Lecture Notes in Artificial Intelligence, pages 1–15. Springer, 2002.
13. U. Hustadt, B. Motik, and U. Sattler. Reasoning for description logics around $\mathcal{SHIQ}$ in a resolution framework. Technical Report 3-8-04/04, June 2004.
14. T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In C. Knoblock and A. Levy, editors, *AAAI Spring Symposium ob Information Gathering from Heterogeneous, Distributed Environments*, 1995.
15. Zsolt Nagy, Gergely Lukácsy, and Péter Szeredi. Translating description logic queries to Prolog. In *Proc. of PADL, Springer LNCS 3819*, pages 168–182, 2006.
16. Interface Definition Language. ISO International Standard, number 14750.

Combining OWL with F-Logic Rules and Defaults

Heiko Kattenstroth, Wolfgang May, and Franz Schenk

Institut für Informatik, Universität Göttingen,
{hkattens|may|schenk}@informatik.uni-goettingen.de

Abstract. We describe the combination of OWL and F-Logic for the architecture of Semantic Web application nodes. The approach has been implemented by combining an existing Jena-based architecture with an external Florid instance. The approach provides a tight language coupling, i.e., the same notions can be defined both by OWL definitions and by F-Logic rules. F-Logic rules are used for e.g., role-value-maps, closed-world-reasoning, (stratified) negation, aggregation, and definition of answer views; additionally the default inheritance of F-Logic can be exploited.

1 Introduction

Description Logics [BCM+03] provide the underlying base for the dominating data model and languages for the Semantic Web, RDF, RDFS, and OWL [OWL04]. It is based on the notion of classes, auch as Person, Country, City, and properties, auch as hasName, hasChild, livesIn, hasCapital. This corresponds to unary and binary relations in First-Order-Logic and relational databases, such as Person(john), hasName(john,"John"), and livesIn(john,berlin). Description Logics additionally allow for further specifications of classes (and properties) that have no direct equivalent in relational databases, e.g. that a parent is a person who has at least one child: $\mathsf{Parent} \equiv \mathsf{Person} \sqcap \exists \mathsf{hasChild}.\top$ or the assertion that children are persons, $\mathsf{Parent} \sqsubseteq \forall \mathsf{hasChild}.\mathsf{Person}$. Such things can be expressed by rules in FOL, but are not inherent concepts of FOL semantics.

Thus, the "built-ins" are more advanced than in FOL, but on the other hand, DL formulas are much restricted to concept expressions. The application of conjunction, and even more disjunction and negation, is only allowed in terms of these built-ins. The Description Logic $\mathcal{SHOIQ}(\mathcal{D})$ [HS05] that forms the base for OWL-DL is decidable, but there are "simple" concepts that are still out of reach in this fragment, e.g., composite properties such as "uncle" as "brother of parent" cannot be expressed.

The combination of Description Logics with rules is thus a prominent research topic, e.g., investigated early in $\mathcal{AL}$-*log* [DLNS91,DLNS98], CARIN [LR96], or more recently in numerous approaches, e.g., DLP [GHVD03], SWRL (earlier: ORL) [HPS04,HPSB+04], DLV/DLVhex [ELST04,EIST06], DL+log [Ros06], OWLFlight [dBLPF05], *DL-safe* rules [MSS05], [Luk07] or [DM07] (see Section 5 for a more detailed analysis).

The goal of our approach is primarily pragmatic: to provide an architecture for an application service node in the Semantic Web (e.g., hosting the information system of an airline service or a university) and providing appropriate interfaces to the outside. For that, we combine Description Logic with (full) F-Logic, which also brings *default inheritance* into play.

DL+Florid provides a *tight* coupling in the sense that the symbols of the DL part and the rule part are not required to be disjoint. Thus, the rules can be used (and are intended) to derive concept memberships and role instances.

The semantics of DL+Florid is defined in a bottom-up way (that is also realized by the implementation that combines the Jena [Jen] Framework with a plugged-in Pellet reasoner [Pel] and Florid 4.0 [FHK+97,FLO06]). The Jena-based DL system provides the core of the architecture that employs Florid as a "slave" for rules and default inheritance. After reviewing the basic notions in Section 2, we describe the architecture and analyze the semantics of our pragmatic approach in Section 3. Inheritance based on defaults is then discussed in Section 4. Section 5 gives a comparison with related work, and Section 6 concludes the paper.

2 Basics

2.1 Overview of Description Logics and RDF/RDFS/OWL

Description Logics (DLs) [BCM+03] are a family of logics for concept reasoning. Their main constructs are classes and properties, expressed by (i) class membership atoms, e.g., $C(a)$ (object a is an instance of class C), property atoms $p(a, b)$ (b is some value of property p of object a), subclass axioms $C \sqsubseteq D$, and class equivalence $C \equiv D$. Different DLs allow or disallow certain constructs for describing class definitions. The current focus is on decidable DLs where complete decision procedures exist. DLs are the underlying framework for the Semantic Web languages RDF, RDFS, and OWL [OWL04] with its variants OWL-Lite, OWL-DL and OWL-Full. For OWL-DL, currently extensions to OWL-1.1 are under discussion. OWL-DL is based on the decidable Description Logic $\mathcal{SHOIQ}(\mathcal{D})$ [HS05]; the extensions belong to $\mathcal{SROIQ}$ which allows additional concepts for specifying properties [HKS06]. As DLs are a restricted fragment of First-Order Logic, FOL model theory and semantics applies to them, which means that in contrast to Logic Programming, *open-world* semantics applies. Reasoners like Pellet [Pel] support OWL-1.1.

2.2 Overview of F-Logic

As stated above, the DL+Florid approach extends DL with deductive rules and default inheritance. Both are features that are natively supported by F-Logic and its implementation in Florid [FHK+97,FLO06]. F-Logic rules are logic programming rules over F-Logic atoms. F-Logic atoms are defined as follows (cf. [KLW95]); we use only properties without parameters (i.e., only the form $o[m \rightarrow v]$, not $o[m@(a_1, \ldots, a_n) \rightarrow v]$).

Definition 1 (Syntax of F-Logic). *The alphabet of an F-Logic language consists of a set $\mathcal{F}$ of* function symbols*, playing the role of object constructors. For convention, function symbols start with lowercase letters whereas variables start with uppercase ones.* Id-terms *are composed from object constructors and variables and are interpreted as elements of the universe.*

In the sequel, let o, c, $d, d_1, \ldots, d_n$, p, $v, v_1, \ldots, v_n$ stand for id-terms or literals. Note that URLs as a subclass of strings can denote objects; e.g.

$$\text{“foo:bla\#john”} : \text{“foo:meta\#Person”}\,[\,\text{“foo:meta\#name”} \rightarrow \text{“John”}\,;\; \text{“foo:meta\#livesIn”} \twoheadrightarrow (\,\text{“geo://de/Berlin”} : \text{“geo:meta\#City”}\,)\,].$$

is a valid F-Logic fragment; see also later examples.

1. *An* is-a atom *is an expression of the form $o : c$ (object o is a member of class c), or $c :: d$ (class c is a subclass of class d).*
2. *The following are* object atoms*:*

2a. $c[p \Rightarrow (d_1, \ldots, d_n)]$ *and* $c[p \Rightarrow\!\!> (d_1, \ldots, d_n)]$*: the values of the* scalar *or* multivalued*, respectively, property p of objects of class c belong (simultaneously) to all classes $d_1, \ldots, d_n$,*

2b. $o[p \rightarrow v]$*: the* scalar *property p of object o has the value v,*

2c. $o[p \twoheadrightarrow \{v_1, \ldots, v_n\}]$*: $\{v_1, \ldots, v_n\}$ are amongst the values of the* multivalued *property p of object o,*

2d. $c[p \bullet\!\!\rightarrow v]$*: for objects of class c, the default value of the* scalar *property p is v.*

2e. $c[p \bullet\!\!\twoheadrightarrow \{v_1, \ldots, v_n\}]$*: for objects of class c, the default values of the* multivalued *property p are $\{v_1, \ldots, v_n\}$.*

An F-Logic rule *is a logic rule $\mathsf{h} \leftarrow \mathsf{b}$ over F-Logic atoms, i.e. is-a assertions and object atoms. An F-Logic* program *is a set of rules.*

The semantics of F-Logic rules and defaults is defined via Herbrand-style structures where the universe consists of ground id-terms. An *H-structure* is a set of ground F-Logic atoms describing an object world, thus it has to satisfy several *closure axioms* related to general object-oriented properties:

Definition 2 (F-Logic Axioms). *A (possibly infinite) set $\mathcal{H}$ of ground atoms is an* H-structure *if the following conditions hold for arbitrary ground id-terms u, $u_0, \ldots, u_n$, and u_m occurring in $\mathcal{H}$:*

- *$u :: u \in \mathcal{H}$ (subclass reflexivity),*
- *if $u_1 :: u_2 \in \mathcal{H}$ and $u_2 :: u_3 \in \mathcal{H}$ then $u_1 :: u_3 \in \mathcal{H}$ (subclass transitivity), analogously, if $u_1 : u_2 \in \mathcal{H}$ and $u_2 :: u_3 \in \mathcal{H}$ then $u_1 : u_3 \in \mathcal{H}$,*
- *if $u_1 :: u_2 \in \mathcal{H}$ and $u_2 :: u_1 \in \mathcal{H}$ then $u_1 = u_2 \in \mathcal{H}$ (subclass acyclicity),*
- *if for ground id-terms u and u' ($u \neq u'$) such that $u_0[u_m \rightsquigarrow u] \in \mathcal{H}$ and $u_0[u_m \rightsquigarrow u'] \in \mathcal{H}$, then $u = u'$, where $\rightsquigarrow$ stands for $\rightarrow$ or $\bullet\!\!\rightarrow$ (uniqueness of scalar properties).*

For a set M of ground atoms, $\mathcal{C}\ell(M)$ denotes the closure of M wrt. the above axioms.

Positive F-Logic programs are evaluated bottom-up by a T_P-like operator including $\mathcal{C}\ell$, providing a minimal model semantics:

Definition 3 (Deductive Fixpoint).
For an F-Logic program P and an H-structure $\mathcal{H}$,

$$\begin{aligned}
T_P(\mathcal{H}) &:= \mathcal{H} \cup \{h \mid (h \leftarrow b_1, \ldots, b_n) \textit{ is a ground instance of some rule of } P \\
&\qquad\qquad \textit{and } b_i \in \mathcal{H} \textit{ for all } i = 1, \ldots, n\} \ , \\
T_P^0(\mathcal{H}) &:= \mathcal{C}\ell(\mathcal{H}) \ , \\
T_P^{i+1}(\mathcal{H}) &:= \mathcal{C}\ell(T_P(T_P^i(\mathcal{H}))) \ , \\
T_P^\omega(\mathcal{H}) &:= \begin{cases} \lim_{i\to\infty} T_P^i(\mathcal{H}) & \textit{if the sequence } T_P^0(\mathcal{H}), T_P^1(\mathcal{H}), \ldots \textit{ converges,} \\ \bot & \textit{otherwise.} \end{cases}
\end{aligned}$$

User-stratified programs are evaluated analogously wrt. Perfect Model semantics. The above semantics that covers deductive rules does not deal with inheritance; this will be described in Section 4.

Correspondence with DL and RDF/RDFS/OWL. In an RDF/OWL setting, objects are identified by URIs and by ids of blank nodes; thus, for theoretical considerations, the restriction to function-free F-Logic is reasonable. Note that in F-Logic, id-terms and objects also stand for properties, in the same way as URIs in RDF. Variables can also occur at arbitrary positions of an atom.

The *isa-atoms* (1) correspond to DL's $C(o)$ and $C \sqsubseteq D$ (i.e., rdf:type and rdfs:subclass). The object atoms (2a) correspond to $C \sqsubseteq \forall p.D$ (i.e., rdfs:range), (2b) and (2c) correspond to the DL property assertions $p(o, v)$ (i.e., the RDF triple (o, p, v)). The inheritance atoms (2d) and (2e) have no equivalent in DL or RDF/OWL. Together with the rules, the semantics of (2d) and (2e) provide the additional expressiveness of DL+Florid.

2.3 Why Rules?

Hybrid approaches that combine DL with deductive rules are of interest for several reasons:

Higher Expressiveness: Positive Rules. Many things that cannot be expressed in OWL can be defined easily with rules, even with often only *positive* rules. These are e.g., composite roles that do not satisfy the *tree property* (although restricted support comes with OWL 1.1), or annotated and computed properties. For instance, connection(city$_1$, city$_2$) is transitive and thus can be expressed in OWL, but as such connections are represented by *role instances*, they cannot have properties like distance (except by reification). Even with reification, it cannot be expressed in OWL that the distance of composite connections is obtained by adding the individual distances (see Example 3 later). Furthermore, aggregations like count, sum, max, avg can be expressed in most rule languages (but these are then actually not just positive rules).

Apart from the above expressiveness issues, logical rules with an operational flavor are often used for ontology integration and data integration.

Higher Expressiveness: Negation under CWA. Another issue is the use of default negation (often also called "negation as failure" which is actually its implementation in Prolog): facts that are not explicitly known are assumed not to hold. This is relevant e.g. when dealing with unmarried or childless people, countries without big cities etc. While "unmarried" is still a property that is often explicitly given, concepts like "country without big cities" are usually to be derived. Default negation also underlies the definition of aggregations, since these implicitly also assume that no additional facts have to be taken into account for the aggregation.

Query Answering. Rules allow for a declarative *and* constructive specification how a result can be obtained. For function-free normal and stratified rules, evaluation of queries (=views) is polynomial.

Example 1 (Rules for Query Answering). Consider the simple geographic ontology of the Mondial database [May07], containing (among others) the notions

Classes: Country, Province, City,
Properties: hasProvince, isProvinceOf, hasCity, cityIn, population.

For some countries, no provinces are known and the cities are directly associated with the countries; for the others, cities are associated with the provinces.

Compute all countries that have at least two cities with more than 1.000.000 inhabitants. In an OWL ontology, a composite relationship between countries and cities covering the hierarchy has to be defined (using transitivity), and concepts (as restrictions) BigCity and CountryWithTwoBigCities must be defined:

```
:isProvinceOf rdfs:subpropertyOf :belongsTo.
:cityIn rdfs:subpropertyOf :belongsTo.
:belongsTo a owl:TransitiveProperty;
    owl:inverseOf :hasProvOrCity.     ## bridge country-prov-city
:Million a owl:DataRange; owl11:onDataRange xsd:int; owl11:minInclusive 1000000.
:HasBigPopulation owl:equivalentClass [a owl:Restriction;
    owl:onProperty :population; owl:someValuesFrom :Million].
:BigCity owl:intersectionOf (:City :HasBigPopulation).
:CountryWithTwoBigCities owl:intersectionOf (mon:Country
   [a owl:Restriction; owl:onProperty :hasProvOrCity;
     owl:minCardinality 2; owl11:onClass :BigCity]).
```

The actual evaluation by a reasoner takes some minutes. In contrast, with (positive) rules, the same can be defined:

```
## (note: use c_Classname for Classes)
Cty:c_BigCity :- Cty:c_City, Cty[population→→Pop], Pop > 1000000.
C[hasBigCity→→Cty] :- C[hasCity→→Cty], Cty:c_BigCity.
C[hasBigCity→→Cty] :- C[hasProvince→→Prov], Prov[hasCity→→Cty], Cty:c_BigCity.
X:c_CountryW2BigCities :- X:c_Country, X[hasBigCity →→{C1,C2}], not C1 = C2.
```

where evaluation is significantly faster. With stratified negation, also e.g. all countries that have no big cities can be listed.

The above example shows that already a "decoupled" combination of an OWL core with rule-based views for computing answers provides certain advantages. Similarly, rules can be used to define concepts although they could be defined as owl:Restrictions, not only as answer views, but also for efficiency.

3 Combining DL+Florid

The goal of the approach is to provide an architecture for an application service node in the Semantic Web (e.g., hosting the information system of an airline service or a university) and providing appropriate interfaces to the outside. Thus, we are primarily interested in a pragmatic approach.

The architecture is shown in Figure 1. The node core is based on the Jena framework [Jen]. It contains a database (e.g. PostgreSQL) as repository for the ontology (i.e., concepts and properties), the facts, and also for the rules. It can be queried with SPARQL [SPQ06] as the current mainstream Semantic Web language. For OWL reasoning, an instance of the Pellet DL reasoner [Pel] is connected to the node. This basic functionality has been extended with a simple update language for RDF data and with support for RDF-level database triggers reacting upon *database update actions*, e.g. to support actual updates when deleting an instance $p(x, y)$ of a property that is symmetric and stored as $p(y, x)$ [MSvL06]. Additionally, a Florid instance is connected as a "slave" for application of F-Logic rules and default reasoning.

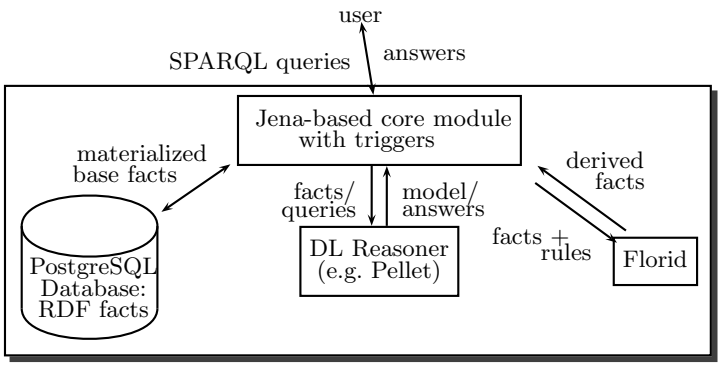


Fig. 1. Architecture of DL+Florid

3.1 Reasoning with DL+Florid

The general idea is to separate concerns into (i) OWL concept (TBox) reasoning, (ii) application of rules, and (iii) inheritance. Reasoning about facts can be

either done in the OWL portion (ABox; e.g., transitivity, inverse, symmetry), or by rules, which allow for much more complex derivations. The knowledge base $KB = (L, P, D)$ is thus partitioned into

- an OWL ontology L,
- a finite set P of rules (in F-Logic or a RuleML-style XML markup), and
- a finite set D of inheritance atoms (defaults).

For things that can alternatively specified by OWL or F-Logic (mostly, class membership characterizations), the *alternating fixpoint* evaluation described below guarantees the same outcome under certain conditions.

This means that usually rule bodies contain only domain notions, and no RDF/RDFS/OWL properties (mainly, to reduce the amount of data to be transferred – using these properties is the native responsibility of the OWL reasoner). Rule *heads* are allowed to contain RDF/RDFS/OWL notions, although this seems to be an unusual case (e.g., used for concept learning).

The evaluation proceeds as follows, taking the OWL ontology in the Jena-based node as starting point:

First Step: Compute the OWL model of a given fact base. Due to its open-world nature, OWL/DL reasoning contains only limited negation. Anything derived *later* will be taken into account as "possible". Doing OWL reasoning first is thus completely safe.

Second Step: Application of deductive rules. All (relevant) facts are exported together with the rules to Florid, where bottom-up-evaluation is applied; the (relevant) resulting facts are sent back to the Jena-based core. In case of positive rules, it is again completely safe wrt. facts derived in later steps by OWL reasoning (see below for a discussion).

Iteration: the above steps are iterated until no new facts can be derived.

Inheritance Step: only when both OWL reasoning and application of rules are not able to derive further facts, default inheritance takes place.

Iteration: As long as new facts have been derived by default inheritance, the above inner iteration is restarted. Note that this corresponds to the F-Logic semantics of default inheritance – applying default inheritance only after the application of rules reached a fixpoint does not derive any new facts, and then restarting the iteration – that has been shown in [MK01] to be "reasonable" and compatible with the Default Logic [Rei80,Poo94] semantics.

3.2 Data Exchange and Handling of URLs

Whereas the evaluation of the OWL specification is based on theory reasoning, the evaluation of rules is based on *ground facts* (and non-ground rules).

Export of facts to Florid: The Jena *model* containing the (base and derived) facts is dumped, and triples are exported as atoms as follows: x rdf:type c where c is an application class (as x_t:c_t), c rdfs:subClassOf d (as c_t::d_t), p rdfs:range

c where p is a non-RDF/RDFS/OWL property (as $\langle\mathsf{owl{:}Thing}\rangle_t[p_t \Rightarrow c_t]$ (if p is known to be functional) or $\langle\mathsf{owl{:}Thing}\rangle_t[p_t \Rightarrow\!\!\!\!\Rightarrow c_t]$ (otherwise)), $x_t\ p_t\ y_t$ where p is a non-RDF/RDFS/OWL property (as $x_t[p_t \rightarrow y_t]$ (if p is known to be functional) or $x_t[p_t \rightarrow\!\!\!\!\rightarrow y_t]$ (otherwise)).

Above, the mapping $*_t$ of the identifiers is as follows: for literals ℓ, i.e., strings and numbers, ℓ_t is the string representation of ℓ, e.g., "bla", 1, or 3.1415. URIs (including ids of blank nodes) are exported as elements of the class url::string, e.g., "http://www.w3.org/2002/07/owl#Thing":url (the distinguished class url::string for URLs has been used for accessing HTML documents in earlier times).

Example 2. For example, the N3 data

```
<foo:meta#Person> a owl:Class.
<foo:meta#name> a owl:FunctionalProperty; a owl:DatatypeProperty.
<foo:meta#livesIn> a owl:FunctionalProperty; a owl:ObjectProperty.
<foo:bla#john> a <foo:meta#Person>; <foo:meta#name> "John";
                  <foo:meta#livesIn> <geo://de/Berlin>.
```

is translated into F-Logic:

```
"foo:meta#Person":url.    "foo:meta#name":url.    "foo:meta#livesIn":url.
"foo:meta#john":url.      "geo://de/Berlin":url.
("foo:bla#john":"foo:meta#Person")["foo:meta#name"→"John";
            "foo:meta#livesIn"↠"geo://de/Berlin"].
```

Export of facts back to Jena: On the way back, the above translation is inverted. Here, only atoms/triples $x{:}c$, $c{::}d$, $x[p \rightarrow y]$, and $x[p \rightarrow\!\!\!\!\rightarrow y]$ are exported where x, c, d, p are members of the class url; y may be an url or a literal. This allows to use auxiliary predicates, classes and identifiers in the rule part that are not exported back to Jena. Note that it is also possible to introduce new classes, properties, and objects by urls in the rule part.

Back in Jena, the returned data is merged with the before data; if new facts have been derived, the alternating process is iterated until a fixpoint is reached.

3.3 Positive Rules

If the program P only consists of positive rules, everything is safe. Independent how many (inner – i.e., between Jena and Rules) iterations are executed until a fixpoint is reached, the resulting structure is a subset (cf. Section 3.5) of what would be derived when using e.g. [ELST04,EIST06].

3.4 Rules with Negation

Florid supports user-defined stratification of programs. Concerning the interference of iterating OWL reasoning and rule application, the CWA of LP negation conflicts with the possibility of later derivation of positive facts by OWL reasoning in the above fixpoint process. The evaluation of stratified programs in

the given alternating combination with OWL reasoning is correct wrt. the *stratified/perfect model semantics* if the extension of predicates that occur negatively in a rule is not changed after evaluating it for the first time (i.e., also not in subsequent OWL rounds). Operationally, this condition can be verified by combining syntactical analysis of rules with runtime monitoring:

- Let Σ^- denote the set of all predicates occurring negatively in a rule body.
- Let $\mathcal{I}$ denote the interpretation after the first iteration of computing the OWL model and applying the rules once (without applying OWL reasoning again). Let $\mathcal{I}_{neg} := \mathcal{I}|_{\Sigma^-}$ ($\mathcal{I}$ restricted to Σ^-).
- For each iteration, check if for the current interpretation $\mathcal{I}'$, $\mathcal{I}'|_{\Sigma^-} = \mathcal{I}_{neg}$.

This allows e.g. to use the negation of all base facts. Since classes in Σ^- often occur in rdsf:range and rdsf:domain axioms, pure ontology analysis will in most cases derive that it is *possible* that they could be extended during reasoning.

An evaluation that is correct in the general case in presence of negation in the rules is only possible in a tightly coupled evaluation in the DL reasoner (which then requires to restrict to DL-safe rules).

Rules for Views and Query Answering. The above condition is trivially satisfied when the predicates in the rule head are disjoint from those used in the OWL part, but occur only in the rule program.

3.5 Pitfalls: Existential Assertions in OWL

In the investigations of hybrid rules, it turned out that the crucial problem are anonymous, implicit objects (see Section 5) that affect decidability (note: these are not the blank nodes, but purely existential objects as in Parent $\equiv$ Person $\sqsubseteq \exists$hasChild.Person, or in Person $\sqsubseteq \exists$hasFather.Person). The constraints on variables developed over time aimed at restricting variables such that they cannot be bound to these anonymous objects; or, taken the other way round [Ros06]: every *head* variable must occur in an LP atoms in the body (such that only objects from the explicit active domain can be bound to variables that occur in the head). Actually, this prevents from deriving anything about these anonymous objects by rules.

Recall that also in SPARQL, even though *reasoning* allows to derive that Joe's father, *jf*, is a person, and also *jf* $\in \exists$hasFather.Person, holds, a query {?X hasFather ?Y} will not yield an answer with ?X/joe (and also not with ?X/*jf*).

When considering such an axiom like Person $\sqsubseteq \exists$ hasFather.Person as a rule, using a function symbol for object invention as usual in F-Logic,

```
father(X):c_Person :- X:c_Person.
```

would create an infinite chain of objects father(father(...(joe))).

From that, the following strategy is a applied for such objects:

Strategy 1 *Don't derive too much about objects that are not explicitly named. Don't use function symbols in rule heads.*

If only **positive** rules are used, missing existential objects can lead to missing data, but not to wrong derivations. E.g. the rule

```
X:c_Uncle :- X:c_Person, X[hasSibling→→Y], Y[hasChild→→Z].
```

with the facts {Person(joe), hasSibling(joe,mary), Parent(mary)}, the latter equivalent with ∃hasChild.Person(mary), would not be sufficient to derive that Joe is an uncle.
If **negative** rules are considered, we have to care for rules like

```
X:c_Childless :- X:c_Person, not X[hasChild→→_Y].
X:c_Fatherless :- X:c_Person, not X[hasFather→→_Y].
```

The first rule would derive c_Childless(X) for all persons, including parents p for whom no *explicit* filler for p.hasChild is known, and the second will derive c_Fatherless(X) for all "bordering" persons of the ontology whose father is only implicitly known.

Strategy 2 *Export all relevant implicitly known "border" objects with their derived properties from the existential axiom from Jena to F-Logic. Then, derive only relevant information about them.*

Which such implicitly known "border" objects are relevant? A border object is relevant, if it is concerned by an atom in some rule. This can be by done inspecting the graph of each rule body (note that e.g.,

```
c_Grandfatherless(X) :- X:c_Person, hasFather(X,_Y), not hasFather(Y,_Z).
```

makes even the grandfathers relevant border objects). Such implicit border objects must/need only be exported if there is no explicit filler for the corresponding role.

Border objects are marked as such, being members of an internal class borderobject. When they occur in a rule head, only relevant properties are actually derived (again based on the graphs of the rules).

Note that this idea is similar to the one presented in [Luk07] that is based on the Herbrand base/model.

3.6 Practical Issues and Additional Functionality for Daily Life

On one hand, the handling of anonymous objects is still "critical", especially in the context of negation. On the other hand, a careful design of the ontology and the rules allow for a reasonable expressiveness obtained by declarative formalisms, which otherwise must be implemented procedurally (and then has

no logical semantics as all). As one of the aims of the approach is to provide a working architecture for individual nodes in the Semantic Web, we went for a pragmatic realization. Some additional functionality that is useful for daily life comes with the use of Florid:

- built-in predicates and operations on strings and numbers,
- creation of URIs (as they are basically also strings – only the membership atom s:url distinguishes them from simple strings).

Example 3 (Train Connections). Consider a train database which contains the "atomic" connections. Assume that for the urls of cities and connections, a globally agreed structure is used. Consider the following fragment (in N3):

```
<travel://db/connections/Hannover-Goettingen> a <travel:meta#Connection>;
  <travel:meta#from> <geo://de/Hannover>; <travel:meta#to> <geo://de/Goettingen>;
  <travel:meta#distance> 120.
<travel://db/connections/Goettingen-Kassel> a <travel:meta#Connection>;
  <travel:meta#from> <geo://de/Goettingen>; <travel:meta#to> <geo://de/Kassel>;
  <travel:meta#distance> 60.
```

and a (simplified) rule that computes composite connections (in F-Logic syntax, where URLs are treated by strings s:url):

```
(U:"travel://meta#Connection")["travel:meta#from"→X; "travel:meta#to"→Z;
        " travel:meta#distance"→D], U:url
:- (C1:"travel://meta#Connection")["travel:meta#from"→X; "travel:meta#to"→Y;
        " travel:meta#distance"→D1],
   (C2:"travel://meta#Connection")["travel:meta#from"→Y; "travel:meta#to"→Z;
        "travel:meta#distance"→D2],
   U = "travel://db/connections/" + X.name + "-" + Y.name, D = D1 + D2.
```

which will generate (already mapped back to N3)

```
<travel://db/connections/Hannover-Kassel> a <travel:meta#connection>;
  <travel:meta#from> <geo://de/Hannover>; <travel:meta#to> <geo://de/Kassel>;
  <travel:meta#distance> 180.
```

Additionally, Florid allows for parsing of HTML and XML data, which can then be transformed by F-Logic rules into RDF to be added to the ontology.

4 Defaults

4.1 Inheritance in F-Logic

Inheritance atoms in F-logic have the form $c[p{\bullet}{\rightarrow}v]$: the class c provides the *inheritable scalar* property p with default value v. For a member $o : c$ without any intermediate class, inheritance results in $o[p{\rightarrow}v]$; for a subclass $d :: c$, inheritance

results in $d[m\bullet\!\!\rightarrow v]$; analogously for multivalued $c[p\bullet\!\!\twoheadrightarrow v]$. Inheritance of defaults is intended to take place, if "nothing else is known".

In [KLW95], *inheritance-canonic* models are defined, based on *inheritance triggers* which extend the above fixpoint semantics: default inheritance is applied after the minimal/stratified model is computed and the rules do not derive any more facts. Objects where for an inheritable property, no value has been derived so far inherit the default, and the minimal/perfect model computation is applied again. Although this definition is formulated in a rather procedural way, we have shown in [MK01] that this semantics is "reasonable" and in most cases compatible with the Default Logic [Rei80,Poo94] semantics. Only when application of rules after applying a default results in an inconsistency, or "attacks" the applicability of the default, a non-supported (wrt. the Default Logic semantics) model results. Below, we show that even this effect is avoided by the DL+Florid architecture where F-Logic reasoning is applied as a "slave" whose results can further be controlled.

4.2 Nonmonotonic Inheritance by Default Logic

In *Default Logic* [Rei80,Poo94], defeasible reasoning is expressed by *defaults*: a default $d = \frac{\alpha : \beta}{w}$ consists of a *precondition* α, a *justification* β and a *consequence* w. Given α, if β can be assumed consistently, one can conclude w. A *default theory* is a pair $\Delta = (D, F)$ where D is a set of defaults and F is a set of formulas.

In an inheritance framework, the superclass condition belongs to α; whereas the checks that inheritance is not preempted by an intermediate class and that the inherited value must be consistent with the knowledge (wrt. the logical rules of the program) fall under β. For characterizing inheritance, only a specialized form of defaults is needed, called *semi-normal defaults* where the precondition $\alpha(\bar{x})$ is a conjunction of atoms, the consequence $w(\bar{x})$ is also an atomic formula, and $\forall \bar{x} : \beta(\bar{x}) \rightarrow w(\bar{x})$ holds. Translating the path-based concept of *inheritance networks*, including avoidance of *decoupling* inheritance in F-Logic syntax can be specified by defaults of the form (cf. [MK01])

$$D_{inh} = \frac{O : C \ , \ C[M\bullet\!\!\rightarrow V] \ : \quad \forall C'((O : C' \wedge C' :: C) \rightarrow C'[M\bullet\!\!\rightarrow V]) \ , \ O[M\rightarrow V]}{O[M\rightarrow V]} \ .$$

The semantics of a default theory is defined in terms of *extensions*. A theory $\mathcal{T}$ is an extension of $\Delta = (D, F)$ if it satisfies certain requirements [Rei80,Poo94,Mak94]; the definitions are non-constructive (where a *quasi-inductive* definition at least gives a guess-and-check characterization).

In our case, the consequent of a default is always a single ground fact. Thus, it is again sufficient to define and analyze the semantics only with the underlying Herbrand Structures, not with theory reasoning.

4.3 Handling Nonmonotonic Inheritance

In the same way as negation, the default reasoning conflicts with the OWA. Here, the conflict is intended: the main reason behind default inheritance is to be able to find a reasonable set of *beliefs* in a situation of incomplete knowledge when any "safe" reasoning –both OWL OWA reasoning and rule-based CWA reasoning– does not lead to additional knowledge. Thus, default inheritance atoms are evaluated in an *outer* iteration. When an alternating fixpoint of OWL and rules is reached, apply one applicable default, and restart the alternating fixpoint.

In our setting, this is accomplished by saving the current model, applying the default, applying OWL and rules reasoning up to the next fixpoint, and if this structure is consistent, continue with it. If a theory has extensions, then this process leads to an extension; otherwise it at least results in a non-extensible structure that is consistent, but contains "non-fired" default instances. Note that the possibility to put the intermediate model aside during the computation controlled by the Jena module allows to accomplish this guarantee in contrast to the F-Logic/Florid-only semantics discussed in [MK01].

In [BH95], it is shown that even for the simple DL $\mathcal{ALCF}$, Reiter's semantics for open defaults leads to undecidability. When considering only individuals that are mentioned explicitly in the ABox, the task becomes decidable. In our approach, the ontology is restricted to explicit facts before submitting it to Florid – thus, the above obviously applies. Note that implicit border objects have to be ignored when checking for applicable defaults.

5 Comparison

While both OWL-DL and function-free Horn rules are decidable fragments of first-order logic, however, the combination leads to undecidability in general. *DLP* [GHVD03] is the, not very expressive, but decidable, intersection of DL and LP. The other end of the spectrum is *SWRL* (earlier: ORL) [HPS04] which is the *union* of DL and LP, which is in general undecidable.

$\mathcal{AL}$*-log* [DLNS91] proposes hybrid rules in a constraint LP style, where the LP Datalog clauses in the body are extended with DL class membership constraints; the heads of the rules may only contain the LP predicates. CARIN [LR96] extends this to allow also DL roles in the Datalog bodies. *Role-safeness*, i.e., in every DL role atom, at least one variable also occurs in a base predicate (i.e., an LP predicate which does not occur in any rule head), guarantees decidability (wrt. $\mathcal{ALCNR}$). A similar strategy is followed in *DL-safe* rules [MSS05], where it is shown that it is sufficient to require *each* variable in the rule to occur in a non-DL-atom in the rule body (wrt. the more expressive $\mathcal{SHOIN}(D)$/OWL-DL); but here DL atoms are also allowed to occur in rule heads. *DL-safety* also makes sure that each variable is bound only to individuals that are *explicitly* known in the ABox.

The DL+log [Ros06] approach provides a tighter integration and shows that a "weak safeness" condition, i.e., every *head* variable must occur in an LP atom

in the body, guarantees decidability. Thus, in contrast to previous approaches, it is allowed to have variables in the body that only appear in DL atoms (thus, the language now covers conjunctive queries over DL). Again, the focus is that the individuals for which something is derived are *explicitly* known (while implicit objects can be significant in the body). Decidability is obtained for all DLs where CQ/UCQ containment is decidable; this includes the logic $\mathcal{DLR}$ [CGL+98], which is weaker than e.g. OWL-DL.

In [MR07], the *MKNF (Minimal Knowledge and Negation as Failure)* [Lif91] idea is applied to DL & LP to obtain a unifying framework which covers e.g. DL+log, SWRL, LP under stable models semantics, and Default Logic with fixed universe. The resulting logic is based on the modal logic S5 and preferential models. Again, *DL-safety* guarantees decidability. The special interaction of open- and closed-world reasoning (DL predicates can also occur under CWA negation) allows to express things that cannot be expressed in other approaches.

There are also some "loose integration" approaches, in the sense that the rules part contains queries to the ontology, again in the style of constraint LP. These do not derive ontology predicates in the head: In DLV/DLVhex (DLV with higher-order and external atoms) [ELST04,EIST06], the DL atoms are "external" to the DLV framework (disjunctive LP) that is based on Answer Set Programming. In a similar way, [DM07] integrates DL atoms into an LP framework (normal logic programs under well-founded semantics).

In contrast, OWL Flight [dBLPF05], reduces the OWL part to what can be expressed with (F-Logic) rules, under CWA, and adds constraints.

In [Luk07], the perspective is changed: the above approaches consider the DL, theory reasoning, perspective. When changing to the perspective of rule-based systems, considering Herbrand structures as a base yields decidability without any syntactic restrictions. The paper proposes a guess-and-check algorithm which is in general in $\mathsf{NExp}^{\mathsf{NP}}$, and has polynomial data complexity for normal or stratified programs in combination with DL-Lite. The approach is the most similar to ours amongst the above ones.

Considering implementations, there is the DLVhex system [ELST04,EIST06], the prototype of [DM07], the upcoming SWRL [HPS04,HPSB+04] support in Pellet [Pel], and KAON2 [KA] which encodes OWL into disjunctive Datalog and allows for DL-safe rules.

6 Conclusion

We have described the combination of (i) OWL-DL, (ii) F-Logic rules, and (iii) default inheritance, consisting of a Jena-based OWL node that employs Florid as a "slave" for evaluating rules. In case that one of the components is empty, the semantics coincides with the usual semantics of the respective formalism:

- If the OWL-DL part of the ontology is empty (or a simple schema+data style database which does not contribute to the reasoning), the resulting system just applies the F-Logic rules and default inheritance in a controlled way, resulting in a safe semantics wrt. the investigations in [MK01].

– If the F-Logic rule part is empty, the system just implements OWL-DL with default inheritance. Again, inheritance is controlled in such a way that it inherits only when the application is consistent, exactly mirroring the semantics from Default Logic given in [Rei80].
– If the predicates that occur in rule heads are disjoint from those of the OWL ontology (i.e., the rule part only queries the ontology, as in [ELST04,EIST06] and [DM07]), the semantics is correct and complete for stratified negation.
– If the predicates that occur in rule *bodies* are disjoint from those of the OWL ontology (i.e., the rule part serves for populating the ontology, e.g., by information extraction from Web pages), the semantics is also correct and complete for stratified negation.

The approach is under implementation in the DL+Florid prototype[1]. It provides a feasible integration of OWL ontologies and rules that is to be seen as a declarative alternative to approaches that implement the same by pure (Java) programming. Its advantages are the declarative specification of the rules that allow for rapid prototyping and flexible adaptation of the rules.

Ongoing and future work is concerned with implementing the inheritance semantics completely and investigating the possibilities to enhance the handling of existential anonymous objects. We expect that syntactical analysis of the rules and the dependency graph in many cases allows for identifying a finite set of anonymous objects and their properties that is sufficient for guaranteeing correctness and completeness for all derived facts about explicitly known objects.

Acknowledgements. This research has been funded by the European Commission within the 6th Framework Programme project REWERSE, no. 506779.

References

[BCM+03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[BH95] F. Baader and B. Hollunder. Embedding Defaults into Terminological Knowledge Representation Formalisms. *J. of Automated Reasoning*, 14:149–180, 1995.

[CGL+98] D. Calvanese, G. D. Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. Description Logic Framework for Information Integration. In *Principles of Knowledge Representation and Reasoning (KR)*, pp. 2–13, 1998.

[dBLPF05] J. de Bruijn, R. Lara, A. Polleres, D. Fensel. OWL DL vs. OWL Flight: conceptual modeling and reasoning for the semantic Web. In *WWW Conf.*, 2005.

[DLNS91] F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. A Hybrid System with Datalog and Concept Languages. In *Trends in Artificial Intelligence; AI*IA'91*, Springer LNCS 549, pp. 88–97, 1991.

[DLNS98] F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. AL-log: Integrating Datalog and Description Logics. *J. Intell. Inf. Syst.*, 10(3):227–252, 1998.

[DM07] W. Drabent and J. Małuszyński. Well-Founded Semantics for Hybrid Rules. In *Web Reasoning and Rule Systems (RR)*, Springer LNCS 4524, pp. 1–15, 2007.

[1] `http://www.semwebtech.org/DLFlorid`

[EIST06] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning. In *Europ. Semantic Web Conf. (ESWC)*, pp. 273–287, 2006.

[ELST04] T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining Answer Set Programming with Description Logics for the Semantic Web. In *Principles of Knowledge Representation and Reasoning (KR)*, pp. 141–151, 2004.

[FHK+97] J. Frohn, R. Himmeröder, P.-T. Kandzia, G. Lausen, and C. Schlepphorst. FLORID: A Prototype for F–Logic. *Intl. Conf. on Data Engineering (ICDE)*, 1997.

[FLO06] Florid Homepage. http://www.informatik.uni-freiburg.de/~dbis/florid/, 2006.

[GHR94] D. M. Gabbay, C. J. Hogger, and J. A. Robinson. *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford Science Publications, 1994.

[GHVD03] B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In *WWW Conf.*, pp. 48–57, 2003.

[HKS06] I. Horrocks, O. Kutz, and U. Sattler. The Even More Irresistible SROIQ. In *Principles of Knowledge Representation and Reasoning (KR)*, pp. 57–67, 2006.

[HPS04] I. Horrocks and P. Patel-Schneider. A Proposal for an OWL Rules Language. In *WWW Conf.*, pp. 723–732, 2004.

[HPSB+04] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.w3.org/Submission/SWRL/, 2004.

[HS05] I. Horrocks and U. Sattler. A Tableau Decision Procedure for SHOIQ(D). In *Intl. Joint Conf. on Artificial Intelligence (IJCAI)*, 2005.

[Jen] Jena: A Java Framework for Semantic Web Appl's. http://jena.sourceforge.net.

[KA] KAON2 – Ontology Management for the Semantic Web. `http://kaon2.semanticweb.org/`.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, 1995.

[Lif91] V. Lifschitz. Nonmonotonic Databases and Epistemic Queries. In *IJCAI*, 1991.

[LR96] A. Y. Levy and M.-C. Rousset. CARIN: A Representation Language Combining Horn Rules and Description Logics. In ECAI, pp. 328–334, 1996.

[Luk07] T. Lukasiewicz. A Novel Combination of Answer Set Programming with Description Logics for the Semantic Web. In *ESWC*, 2007.

[Mak94] D. Makinson. General Patterns in Nonmonotonic Reasoning. In [GHR94].

[May07] W. May. The MONDIAL Database, 1999–2007. http://dbis.informatik.uni-goettingen.de/Mondial/.

[MK01] W. May and P.-T. Kandzia. Nonmonotonic Inheritance in Object-Oriented Deductive Database Languages. *J. of Logic and Computation*, 11(4), 2001.

[MR07] B. Motik and R. Rosati. A Faithful Integration of Description Logics with Logic Programming. In *Intl. Joint Conf. on Artificial Intelligence (IJCAI)*, 2007.

[MSS05] B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with rules. *J. of Web Semantics*, 3(1):41–60, 2005.

[MSvL06] W. May, F. Schenk, and E. von Lienen. Extending an OWL Web Node with Reactive Behavior. In *Principles and Practice of Semantic Web Reasoning (PPSWR)*, Springer LNCS 4187, pp. 134–148, 2006.

[OWL04] OWL Web Ontology Language. http://www.w3.org/TR/owl-features/, 2004.

[Pel] Pellet: An OWL DL Reasoner. http://pellet.owldl.com.

[Poo94] D. Poole. Default Logic. In [GHR94].

[Rei80] R. Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 12(1,2), 1980.

[Ros06] R. Rosati. DL+log: Tight Integration of Description Logics and Disjunctive Datalog. In *Principles of Knowledge Representation and Reasoning (KR)*, 2006.

[SPQ06] SPARQL Query Language for RDF. www.w3.org/TR/rdf-sparql-query/, 2006.

HD-rules: a hybrid system interfacing Prolog with DL-reasoners

Włodzimierz Drabent[1,3], Jakob Henriksson[2], and Jan Małuszyński[3]

[1] Institute of Computer Science, Polish Academy of Sciences,
ul. Ordona 21, Pl – 01-237 Warszawa, Poland
drabent@ipipan.waw.pl
[2] Fakultät für Informatik, Technische Universität Dresden
jakob.henriksson@tu-dresden.de
[3] Department of Computer and Information Science,
Linköping University, S 581 83 Linköping, Sweden
janma@ida.liu.se

Abstract. The paper presents a prototype system *HD-Rules (Hybrid integration of Description Logic and Rules)* that integrates normal clauses under the well-founded semantics with ontologies specified in Description Logics. The system is hybrid: it re-uses XSB Prolog for rule reasoning and existing OWL reasoners for ontology reasoning. This makes it possible to use some Prolog built-ins (like arithmetic) in the rules. The system itself is written in XSB Prolog; its interface to OWL employs Java. The paper outlines the principles of the integration, illustrates the use of the system on examples, and discusses in detail the main implementation techniques.

1 Introduction

This paper presents a prototype system integrating Description Logic reasoners compatible with the DIG-standard with normal clauses as used in logic programming. The work is based on the well-founded semantics of logic programs and on the ideas of constructive negation in logic programming, as discussed in [6]. The prototype implements a language of *hybrid rules* that extends normal clauses. The hybrid rules allow queries to OWL ontologies in their bodies. Prolog arithmetic and some other Prolog built-ins can also be used.

Integration of rules and ontologies is presently addressed by many researchers as a necessary step in extending Semantic Web technology. The Web Ontology Language OWL, standardized by W3C is supported by several reasoners, while there is yet no common agreement about the rule level. While the main variant of OWL is based on DL (Description Logics), hence on FOL (first order logic), it is often claimed that rules should allow non-monotonic reasoning. Non-monotonic reasoning has been investigated within logic programming. The two main kinds of the semantics proposed are the Answer Set Semantics (Stable Model Semantics) and the well-founded semantics. The well-known proposals for integration of rules and ontologies [7, 13] are based on Answer Set Semantics. The well-founded semantics is used in [8] but in a way different from our approach (for more detailed discussion see [6]).

Important aspects of the work presented in this paper are

- Our approach is based on the well-founded semantics of normal programs, and is compatible with FOL: if the non-monotonic negation is not used in the rules, the answers to queries are logical consequences of the set of FOL axioms consisting of the rules and of the ontology.
- We allow the use of term constructors and some Prolog built-in predicates (e.g. arithmetic) in the hybrid rules.
- The approach makes it possible to re-use existing reasoners (for DL, and for Prolog with the well-founded semantics). This substantially simplifies its implementation.
- We explain in detail the principles of implementation.

This paper extends the short paper [5] in the following ways. The hybrid rule language of [5] is extended by allowing term constructors (e.g. Prolog list constructors) and some Prolog built-in predicates. The main implementation issues are discussed in more detail. In particular we describe how hybrid rules are compiled into Prolog, and how the ontology queries in the rules are processed.

A declarative semantics of hybrid programs was defined in our previous work [6] and is briefly summarized in Section 2. Its main idea is that a Herbrand model of a hybrid program is constructed for every model of the underlying ontology. This is similar to the notion of NM-model in [13]. The latter is however based on on the notion of stable model, while our construction uses the notion of well-founded model. Our implementation is based on the operational semantics of [6], which answers queries by combining a constructive negation approach to SLS-resolution [4] with ontological reasoning. The operational semantics is sound wrt. the declarative one and complete for a restricted class of hybrid programs (see [6] for details).

2 Hybrid Programs

In this section we first introduce the syntax of hybrid programs and provide an example program. We then briefly discuss the declarative semantics of hybrid programs and its operational semantics. We conclude with some more examples.

The Syntax. The syntax of hybrid programs is derived from the syntax of the component languages. The component languages considered here are the language of normal logic programs, and some DL-based ontology language. We assume that the alphabets of predicate letters of logic programs and of the ontology language are disjoint, but both languages have common variables and constants. (The alphabet of logic programs also includes function symbols of non zero arity.) Literals, atoms and predicate symbols of logic programming will be called, respectively rule literals, rule atoms, etc. A standard logic programming syntax is extended by allowing ontological constraints to appear in the rule bodies. Thus, a hybrid rule looks as follows:

$$R_0 :- R_1, \ldots, R_k, neg(R_{k+1}), \ldots, neg(R_n), dl(C_1), \ldots, dl(C_m).$$

where $R_0, R_1, \ldots, R_n$ are rule literals and $C_1, \ldots, C_m$ are constraints. At the moment we only allow here constraints of the form $C(x)$ or $\neg C(x)$ where C is a concept of the ontology and x is a variable or a constant. A hybrid program is a pair (T, P) where T is an ontology (a finite set of axioms of a DL) and P is a finite set of hybrid rules with constraints over the alphabet of T. In practice T will be provided by a declaration associating a short name (prefix) with the URI of the ontology. This is here done by using the syntax `use` *'ontology_uri'* `as` *'prefix'*. Any predicate symbol p from the ontology is represented in the hybrid rules as `prefix#`p.

Example 1. Consider a program consisting of the set of hybrid rules P shown in Listing 1.1, and an ontology

$$\mathit{Finland} \sqsubseteq \mathit{Europe}.$$

(A T-box of one axiom and an empty A-box).

```
use 'http://dev.metajungle.info/owl/geography.owl' as 'g'.

win(X) :- move(X,Y), neg(win(Y)).

move(e,f) :- dl(g#Europe(f)).
move(c,f) :- dl(neg(g#Finland(f))).

move(b,a).  move(a,b).  move(a,c).  move(c,d).  move(d,e).
```

Listing 1.1. An example hybrid program describing a two-person game.

The hybrid program in Listing 1.1 describes a two-person game, where each of the players, in order, moves a token from a node of a directed graph

$$\begin{array}{rcl} & d \rightarrow e & \\ & \uparrow \quad \Downarrow & \\ b \leftrightarrow a \rightarrow c & \Rightarrow & f \end{array}$$

over an edge of the graph. The nodes correspond to geographical objects specified in an ontology (e.g. cities) and are represented by constants. Some edges of a graph (represented in the example by the *move* facts) are labelled by constraints (added as constraints to the respective facts). The constraints refer to the ontology. A move from a position x to a position y is enabled if there is an edge from x to y and the constraint is satisfied. The predicate $win/1$ characterizes the winning positions of the game, as described below.

A position is winning if a move is enabled to a position which is not winning (call it losing). Obviously a position where no moves are enabled is losing. Thus, position f is losing. The move from e to f is enabled only if f is in Europe. This cannot be concluded from the ontology. Consequently we cannot conclude that e is a winning position. Similarly, we cannot conclude that f is not in Finland which is required for the move from c to f. However, it follows from the ontology that if f is not in Europe it is also not in Finland. Hence one of the conditions holds for f. Consequently c is a winning position: if f is in Europe, e is winning, d is losing and c is winning. Otherwise f is not in Finland and c is winning.

The positions a and b cannot be classified as winning or losing, since from a one can always move to b where the only enabled move is back to a. The third logical value *undefined* is assigned to $win(a)$ and $win(b)$. The status of d and e is also not clear, but for different reasons discussed above. In some, but not all models of the ontology e is winning and d is losing and in the remaining ones the opposite holds.

The Declarative Semantics. In [6] we define a formal semantics of hybrid programs, extending the well-founded semantics of normal programs. Here we survey informally the main ideas. The well-founded semantics of normal programs is three-valued and gives a fixpoint formalization of the way of reasoning illustrated by the game example, when the constraints are neglected. It assigns to every element of the Herbrand base one of the logical values *true* (e.g. $win(c)$), *false* (e.g. $win(f)$) or *undefined* (e.g. $win(a)$).

The constraints added to the rule bodies refer to the ontology. As illustrated by the example, a ground instance of a constraint may have different truth values in different models of the ontology. Consider a hybrid program (T,P) (where T is a set of first order axioms, and P a set of hybrid rules), a model M of T, and the set $ground(P)$ of all ground instances of the rules in P. Each of the ground constraints is either true or false in M. Denote by P/M the set obtained from $ground(P)$ by removing each rule including a constraint false in M and by removing all constraints (which are thus true) from the remaining rules. As P/M is a normal program it has a standard well-founded model. A ground literal p (or $neg(p)$) is said to follow from the program iff p is true (respectively p is false) in the well-founded model of P/M for every M. The declarative semantics of P is defined as the set of all ground literals which follow from the program. Notice that there may be cases where neither p nor $neg(p)$ follows from the program. This happens if there exist models M_1 and M_2 of T such that the logical values of p in the well-founded models of P/M_1 and P/M_2 are different, or if the logical value of p in the well-founded model of P/M is *undefined* for every model M of T.

Notice that the semantics involves two kinds of negation: the monotonic negation of the ontology ($\neg$) and the non-monotonic negation (*neg*) of the well-founded semantics. The former is applicable only to ontology predicates, the latter only to rule predicates. Thus in our implementation we can denote both by the same symbol (*neg*).

The Operational Semantics. The implementation discussed below focuses on answering atomic queries and ground negated literal queries. We now informally sketch the principles of computing answers underlying our implementation. They are based on the operational semantics of hybrid programs presented in [6] by abstract notions of two kinds of derivation trees, called *t-tree* and *tu-tree*, which are defined by a mutually recursive definition. These notions extend the well-known concept of SLD-trees to the case of hybrid programs, to handle negation and constraints. In the presentation below the term *derivation tree (d-tree)* is used whenever the statement applies to both kinds of trees.

The nodes of d-trees are labelled by goals, consisting of rule literals and constraints. The conjunction of all constraints of a node will be called *the constraint of the node*. The label of the root is called *the initial goal* of the tree. A leaf of a d-tree is called *successful* if it does not include rule literals and if its constraint is satisfiable. The other

leaf nodes are called *failed* leaves. In every node containing rule literals, one of them is distinguished as the *selected literal* of the node. As usual, we assume existence of a selection function that determines the selected literals of the nodes.

In the case when the initial goal g of a d-tree is ground the tree has the following property. Let $C_1, \ldots, C_k$ be the constraints of all successful leaves of a d-tree t. Then:

- If t is a t-tree then $(\exists(C_1 \vee \ldots \vee C_k)) \rightarrow g$. Thus g follows from the program if $\exists(C_1 \vee \ldots \vee C_k)$ is a logical consequence of the ontology.
- If t is a tu-tree then $(\neg\exists(C_1 \vee \ldots \vee C_k)) \rightarrow \neg g$. Thus the negation of g follows from the program if $\neg\exists(C_1 \vee \ldots \vee C_k)$ (or equivalently $\neg\exists C_1 \wedge \ldots \wedge \neg\exists C_k$) is a logical consequence of the ontology.

Thus to answer a ground query g our prototype constructs a t-tree with g as its initial goal and checks if the respective disjunctive constraint, existentially quantified, is a logical consequence of the ontology. If it is then g is true (in the declarative semantics of the program).

If g is not ground and C_i is (the constraint of) a successful leaf of a t-tree for g then $\exists C_i \rightarrow g\theta$ follows from the program, where θ is the composition of the mgu's along the branch from g to C_i, and the quantification is over those variables that do not occur free in $g\theta$. Again, if $\exists C_i$ is a logical consequence of the ontology then $g\theta$ follows from the program.

We now explain how d-trees are constructed for a given initial goal g . This is similar to construction of an SLD-tree. Every step is an attempt to extend a tree which initially has only one node labelled by g. At every step one node n, not marked as failed, is considered. Let q be the goal of the node, let s be its selected literal and let C be the conjunction of its constraints. The following cases are considered separately:

1. *s is positive*. For each rule of the program, for which there exists a variant h :- B, Q of the rule such that
   - s and h are unifiable with a most general unifier θ, and
   - the constraint $(C \wedge Q)\theta$ is satisfiable,

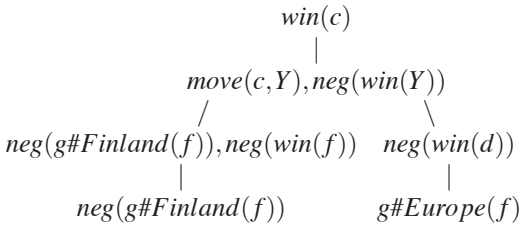
   a child is added to n with the label obtained from $q\theta$ by replacing s by $(B, Q)\theta$. If no such rule exists then n is marked as a failed node.
2. *s is negative*, i.e. of the form $neg(l)$. Two sub-cases are:
   (a) If l is non-ground, or recursion through negation has been discovered (see below) then:
      - If the d-tree is a t-tree then the node n is marked as a failed node and won't be considered in the next steps of the derivation.
      - If the d-tree is a tu-tree then a child is added to n with the label obtained be removing s from q.

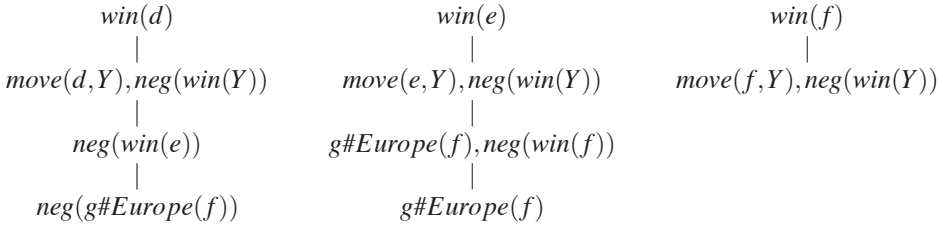
   (b) Otherwise l is ground; the step is completed after construction of a separate d-tree t for l. The kind of the separately constructed tree is different from the kind of the current tree, thus it is a tu-tree if the latter is a t-tree, and t-tree if the latter is a tu-tree. Let $C_1, \ldots, C_k$ be the constraints of the successful leaves of t. If the constraint $C' = C \wedge \neg\exists C_1 \wedge \ldots \wedge \neg\exists C_k$ is satisfiable then a child is added to node n with the label obtained from q by removing s and replacing C by C'.

Otherwise the node is marked as failed. In particular, if $k = 0$ (no successful leaf) C' is equivalent to C. On the other hand, if some $C_i (1 \leq i \leq k)$ is *true*, the constraint C' is equivalent to *false* and is not satisfiable.

For more details, see [6]. In general the construction of a d-tree may not terminate for recursive rules. Recursion not involving negative literals may produce infinite branches of the constructed d-tree. Recursion through negation may require construction of infinite number of d-trees. In our implementation tabling is used; it allows to cut the loops in the case when the same goal re-appears in the process.

Example 2. When a goal $win(c)$ is given to the program from Example 1 then a t-tree for $win(c)$, tu-trees for $win(f)$ and $win(d)$, and a t-tree for $win(e)$ are constructed:





Notice that the leaf of the tu-tree for $win(f)$ is failed (and the leaves of the other trees are successful). The disjunction $\neg g\#Finland(f) \vee g\#Europe(f)$ of the successful leaves of the t-tree for $win(c)$ is found to be a logical consequence of the ontology. Hence the answer for $win(c)$ is Yes.

Notice that for the goals above there is no difference between t- and tu-trees, as the case 2a is not involved.

Let us now consider a t-tree for $win(X)$. The root $win(X)$ has one child $move(X,Y),neg(win(Y))$, which in turn has 7 children, one per each clause for *move*. Three of the children are failed leaves: $neg(win(a))$, $neg(win(b))$, $neg(win(c))$; the corresponding substitutions bind X to b,a,a respectively. The first two nodes are failed due to infinite recursion through negation; $neg(win(c))$ is failed as the constraint $\neg\neg g\#Finland(f) \wedge \neg g\#Europe(f)$ obtained from a tu-tree for $win(c)$ is unsatisfiable.

The remaining four children lead to success leaves. (The corresponding subtrees occur in the trees above.) The leaves and the corresponding substitutions for X are:

$g\#Europe(f)$	$\neg g\#Finland(f)$	$g\#Europe(f)$	$neg(g\#Europe(f))$
$\{X/e\}$	$\{X/c\}$	$\{X/c\}$	$\{X/d\}$

The answers for query $win(X)$ are: $X = e$ provided that $g\#Europe(f)$ (obtained from the first leaf), $X = d$ provided that $\neg g\#Europe(f)$ (obtained from the last leaf), and $X = c$ (as the disjunction of the leaves with substitution $\{X/c\}$ is a logical consequence of the ontology).

In our presentation above, we imposed certain restrictions on the operational semantics from [6]. 1) We deal only with ground negated goals; for non ground ones only a crude, but sound, approximation is used (case 2a). This is to avoid (in)equational constraints in the goals of d-trees; dealing with such constraints would be rather complicated. 2) We construct all the successful leaves of a tu-tree, while in general the constraints of any cross-section of the tree could be taken instead. Choosing the successful leaves as the selected cross-section produces a most general result. (Formally, the constraint C' from case 2b is the most general among those that could be obtained from the given tu-tree for l.) On the other hand, this approach fails if the set of the leaves is infinite. (More precisely, if the set of the constraints of the leaves, up to variable renaming, is infinite.) In such a case, choosing some finite cross-section can provide useful results. In the current work we prefer the simplicity of the restricted solution to the power of the general one. 3) A simplification of the operational semantics from [6] is that when a literal $neg(l)$ is selected in a goal q (case 2b above), the root for a new d-tree is l. (The constraint of q is not passed to the new tree.) This usually results in smaller constraints of the goals in d-trees, and in simpler and more powerful tabulation of infinite sequences of d-trees.

In practice it may be too expensive to check satisfiability of the constraint of each goal. Thus the trees constructed by an actual implementation may contain more nodes and have some additional success leaves, however with unsatisfiable constraints. Clearly this does not violate the soundness of the operational semantics.

Further examples.

Example 3 (A non Datalog program). Here an additional requirement to the game from the previous example is added. Each node can be visited at most once. The list of forbidden nodes is kept in the second argument of predicate $win/2$.

```
use 'http://dev.metajungle.info/owl/geography.owl' as 'g'.

win(X) :- win(X,[]).

win(X,History) :- move(X,Y,History), neg(win(Y,[X|History])).

move(A,B,History) :- edge(A,B), neg(member(B,History)).

edge(e,f) :- dl(g#Europe(f)).
edge(c,f) :- dl(neg(g#Finland(f))).
edge(b,a).      edge(a,b).      edge(a,c).
edge(c,d).      edge(d,e).

member(X,[X|T]).
member(X,[H|T]) :- member(X,T).
```

Prolog built-in predicates can be used in hybrid rules. In principle, any built-in predicates without side-effects (like modifying the program itself, referring to files, etc) can

be used. The semantics of built-in predicates is the same as in Prolog. In particular, invocations of arithmetic predicates have to satisfy the relevant groundness requirements. As the implementation employs the Prolog selection rule, the programmer's reasoning about the form of predicate invocation arguments is the same as for Prolog programs.

As many built-ins, like $var/1$ do not have any declarative semantics, we suggest that only such built-in predicates are used, for which if an atom A fails (succeeds instantiated to $A\theta$) then each instance of A fails (respectively succeeds instantiated to an instance of $A\theta$).

Example 4 (Using Prolog built-ins). Here the additional condition is changed, so that for each node a number of allowed visits is given. An atom $membern(X,L,N)$ is true iff element X occurs N times in list L. Prolog arithmetic is used to deal with integers (built-in predicates $is/2$ and $</2$). Also the built-in \=/2 (non-unifiability check) is employed to check disequality of nodes. (This could be done without built-ins, by replacing E\=G with $neg(eq(E,G))$, and defining $eq/2$ by $eq(X,X)$.)

```
use 'http://dev.metajungle.info/owl/geography.owl' as 'g'.

win(X) :- win(X,[]).

win(X,History) :- move(X,Y,History), neg(win(Y,[X|History])).

move(A,B,History) :- edge(A,B), restriction(B,R), membern(B,History,N), N<R.

edge(e,f) :- dl(g#Europe(f)).
edge(c,f) :- dl(neg(g#Finland(f))).
edge(b,a).     edge(a,b).     edge(a,c).
edge(c,d).     edge(d,e).

restriction(a,7).     restriction(b,6).     restriction(c,1).
restriction(d,1).     restriction(e,1).     restriction(f,1).

membern(E,[],0).
membern(E,[E|L],N1) :- membern(E,L,N), N1 is N+1.
membern(E,[G|L],N)  :- E\=G, membern(E,L,N).
```

Notice that, in contrary to Example 1, infinite games are impossible in the last two examples. Hence each position is either winning, or losing (i.e. the value of $win(X,History)$ is either *true* or *false*, for any node X and list *History*).

3 The prototype

This section presents a concrete prototype implementing the operational semantics presented in Section 2. We present a general architecture of the system, describe compilation of hybrid programs and queries into Prolog, explain the usage of tabulation to prune infinite computations, and present how description logic constraints are dealt with.

Figure 1 shows the user interface of the prototype. The user has entered the program from Example 1 and a query into the respective fields. Pressing the "Query" button compiles the program and the query, and then produces an answer to the query. The "Compile" button displays the compiled program. The prototype is under construction, its current version is available at http://www.ida.liu.se/hswrl/.

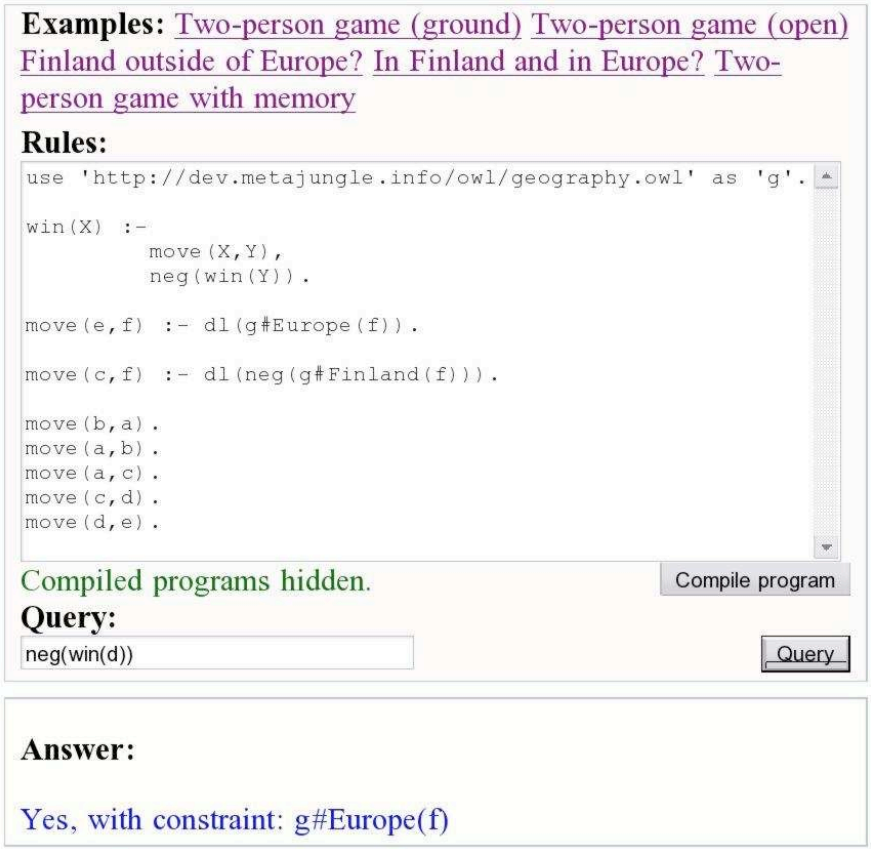


Fig. 1. The web-interface of the hybrid reasoner answering a query with a constrained answer.

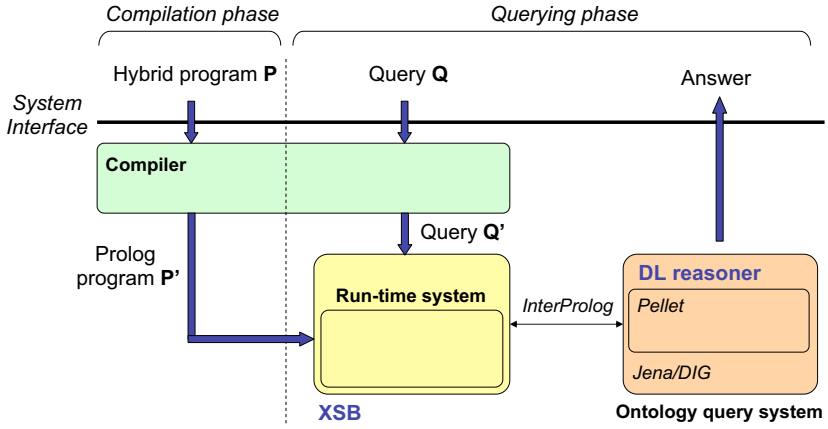


Fig. 2. Prototype architecture overview.

General architecture. An overview of the main components of the reasoning system is shown in Figure 2. The systems is comprised of three main components:

1. *Compiler.* In order to reuse a Prolog engine for handling the rule part of a hybrid knowledge base, we compile hybrid rules (and queries) to plain Prolog.
2. *Run-time system.* When querying a hybrid program, the reasoner queries the compiled program (using a compiled query). The run-time system is implemented in Prolog. It is responsible for constructing derivation trees and for proper handling of constraints, as they appear in the underlying hybrid program.
3. *Ontology query system.* The run-time system interactively communicates with an ontology query system, responsible for checking ontological constraints.

Both the run-time and ontology query systems treat the underlying Prolog and DL engines as black boxes. No modifications of the engines are needed; in principle any Prolog implementation supporting communication with Java, and any DL reasoner with a DIG interface may be used. It is desirable that the Prolog engine provides tabulation, which discovers (some) infinite branches of search trees. Otherwise a rather poor approximation of the well-founded semantics is obtained. In our prototype we use XSB Prolog system [14] and Pellet [12].

Before discussing the main system components in detail, we motivate the use of protocols and API's that we depend upon for the realization of the system.

InterProlog [11] is a Prolog-Java interface, enabling communication and data sharing between Prolog and Java programs. Communication can be handled both ways, that is, passing Java objects to Prolog and sending Prolog terms to Java programs. There is no standard interface between Prolog systems and DL-reasoners. However, there are API's for handling communication with DL reasoners from Java programs (e.g. Jena [10]). Thus, communicating with Java programs from Prolog enables access to DL-reasoners from Prolog.

Two Prolog predicates are provided by InterProlog to aid in communication with a Java program. First, in order to prepare for the passing of data between Java and Prolog, InterProlog provides the predicate `buildTermModel/2`. This predicate encodes Prolog terms, such that they might be sent to a Java program and be properly understood using the Java API provided by InterProlog. E.g. `buildTermModel([1,2,3],P)` succeeds with the variable `P` unified with the encoding of the list `[1,2,3]`. Second, the predicate `javaMessage/3` is provided to invoke a specific Java method and thereby enabling the passing of prepared Prolog terms as arguments. E.g. the Prolog goal `javaMessage('Class'-obj,R,method(P))` produces a result `R` of calling the Java method `Class.obj.method(P)`.

A protocol for communication with DL-reasoners is provided by DIG and is emerging as a standard [3]. The implementation does not directly use DIG, but the DL-reasoner interface provided by Jena [10] employs DIG. Thus, as long as a DL-reasoner is DIG-compliant, it may be plugged into our system.

Compiling HD rules into XSB Prolog. The hybrid rules include DL constraints and cannot be directly used in Prolog computations. Each negative literal encountered in a

Prolog computation initiates construction of an underlying derivation tree, where DL-constraints also have be handled. To address these issues a given HD-Program is first compiled into a Prolog program. We here explain the idea of the compilation and discuss the details.

The underlying idea of the compilation technique is to prevent the constraints to be selected by the Prolog selection function during rule execution. However, since constraints may share variables with rule predicates, such constraint variables need to be processed and unified when the corresponding variables in the rule predicates are. Achieving this is possible by moving the constraint predicates into arguments of other predicates (which are selected by the selection function). In general, each n-ary non-constraint predicate is extended with three additional arguments during compilation (where $\longrightarrow$ represents the compilation step):

$$p(\bar{u}) \;\longrightarrow\; p(\bar{u}, Table, Constraint, Mode)$$

The first extra argument (*Table*) is used to prevent infinite recursion through negation (further explained below). The second argument (*Constraint*) will represent the constraints accumulated during resolving the sub-goal $p(\bar{u})$. The third argument (*Mode*) will obtain a value `t` or `tu`, depending on which kind of derivation tree is currently being constructed. While compiling a clause, the *Constraint* argument for each literal is a unique variable. On the other hand, the *Table* and the *Mode* argument are each the same variable for all the rule literals of the clause (including the head literal).

When a negative literal $neg(p(\bar{u}))$ is encountered, a new derivation tree is to be constructed for the positive version $p(\bar{u})$ of the literal, and the constraints accumulated along the branches of the tree are to be treated as described in Section 2. This is done by a predicate $negation/4$. Thus negative rule literals are compiled into appropriate invocations of this predicate:

$$neg(p(\bar{u})) \;\longrightarrow\; negation(p(\bar{u}), Table, Constraint, Mode)$$

Let $t(R)$ denote a rule literal R translated as described above. A hybrid rule

$$R_0 \;:-\; R_1, \ldots, R_n, dl(C_1), \ldots, dl(C_m)$$

is compiled into

$$\begin{aligned} t(R_0) :-\; & t(R_1), \ldots, t(R_n), \\ & andAppend(Constraint_1, \ldots, Constraint_n, C_1, \ldots, C_m, Constraint_0) \end{aligned}$$

where $Constraint_i$ is the second additional argument of $t(R_i)$ (for $i = 0, \ldots, n$). The predicate *andAppend* unifies $Constraint_0$ with the conjunction of the constraints of the rule and the constraints accumulated by the invocations of $t(R_1), \ldots, t(R_n)$. In practice this is not a single atom, but $n-1$ atoms with a predicate $andAppend/3$; they include a term which represents the conjunction of $C_1, \ldots, C_m$. (The constraints are represented as conjunctions, more precisely as lists built with symbols $and/2$ and $true/0$; predicate $andAppend/3$ joins two such lists.) If $n < 2$ then *andAppend* is not used. Instead, $Constraints_0$ in the head is replaced by a term representing the conjunction of $C_1, \ldots, C_m$ when $n = 0$ (or the conjunction of $C_1, \ldots, C_m$ and $Constraint_1$ when $n = 1$).

Predicate $negation/4$ is a main predicate of the run-time system. It constructs a d-tree for its first argument, employing *findall/3* of Prolog. The tree is a tu-tree if the *Mode* argument is t, and a t-tree otherwise. Moreover, $negation/4$ collects the constraints $C_1,\ldots,C_k$ of the success leaves of the tree, and returns in its third argument the formula $\neg\exists C_1 \wedge \ldots \wedge \neg\exists C_k$. (If some C_i is *true* then $negation/4$ fails, as in such case $\neg\exists C_1 \wedge \ldots \wedge \neg\exists C_k$ is unsatisfiable.) If the tu-tree cannot be constructed (due to non ground root or infinite recursion through negation) then $negation/4$ returns *true* or fails, according to case 2a of the description of the operational semantics.

Hybrid rules may contain Prolog built-ins. Literals with built-in predicates are passed unchanged to the compiled program, without adding the three extra arguments. If such literal is negative then, in the current version of the system, the negation is converted into Prolog negation as failure.

Compiling queries. Queries to hybrid programs must also be compiled before queried wrt. the compiled hybrid program. Queries consisting of a single literal are compiled in the following way:

$$p(\bar{u}) \longrightarrow p(\bar{u},[],Constraint,t)$$
$$neg(p(\bar{u})) \longrightarrow negation(p(\bar{u}),[],Constraint,t)$$

That is, the tabling table is initially empty (the empty list), the constraints will be collected in a variable (here *Constraint*), and the top level d-tree to be constructed is a t-tree. (For a negative literal this tree consists of two or three nodes only.)

Each answer for a compiled query provides a constraint $Constraint\theta$, and an instance $\bar{u}\theta$ of the variables of the original query. If the constraint is unsatisfiable w.r.t. the ontology, the answer is discarded. If the constraint is a logical consequence of the ontology, then $p(\bar{u}\theta)$ follows from the hybrid program.[4] Otherwise, implication $Constraint\theta \rightarrow p(\bar{u}\theta)$ follows from the program.

If there are many answers $Constraint\theta_1,\ldots,Constraint\theta_k$ and $p(\bar{u})$ is ground then $Constraint\theta_1 \vee \cdots \vee Constraint\theta_k$ implies $p(\bar{u})$, and the constraint $Constraint\theta_1 \vee \cdots \vee Constraint\theta_k$ is checked w.r.t. the ontology. For a non ground query we can deal similarly with such answers $Constraint\theta_1,\ldots,Constraint\theta_k$ for which the corresponding instances of the goal are the same: $\bar{u}\theta_1 = \cdots = \bar{u}\theta_k$.

Queries that are conjunctions of literals can be compiled similarly to the bodies of hybrid rules; the difference is that [] is used instead of the variable *Table* and t instead of *Mode*.

Example 5. The rule

$$move(A,B,History) :- edge(A,B), restriction(B,R), membern(E,History,N), N<R.$$

from Example 4 is compiled into

```
move( A, B, History, Tbl, Cnst, M ) :-
    edge( A, B, Tbl, Cnst1, M),                andAppend( Cnst1, Cnst23, Cnst ),
    restriction( B, R, Tbl, Cnst2, M ),        andAppend( Cnst2, Cnst3, Cnst23 ),
    membern( E, History, N, Tbl, Cnst3, M ),
    N < R.
```

[4] More generally, it is sufficient that $\exists Constraint\theta$ is a logical consequence, where the quantification is over those free variables of $Constraint\theta$ that do not occur in $p(\bar{u}\theta)$.

Keeping the related compiler predicate simple resulted in a maybe not natural way of placing *andAppend*/3 atoms in the compiled clauses.

The set of hybrid rules of Example 1 is compiled into:

```
win( X, Tbl, Cnst, M ) :- move( X, Y, Tbl, Cnst1, M ),
                          andAppend( Cnst1, Cnst2, Cnst ),
                          negation( win(Y), Tbl, Cnst2, M ).

move( e, f, Tbl, and('g#Europe'(f),true), M ).
move( c, f, Tbl, and(neg('g#Finland'(f)),true), M ).

move( b, a, Tbl, true, M ).      move( a, b, Tbl, true, M ).
move( a, c, Tbl, true, M ).      move( c, d, Tbl, true, M ).
move( d, e, Tbl, true, M ).
```

A query $win(e)$ is compiled into $win(e,[],Cnst,t)$. Executing the latter goal results in calling $negation(win(f),[],Cnst2,t)$, and construction of a tu-tree for $win(f)$ without successful leaves (see Ex. 2). We obtain $Cnst2 = true$ and the initial goal succeeds once, with $Cnst$ bound to $and(g\#Europe(f),true)$ (which is equivalent to $g\#Europe(f)$). This constraint is found to be satisfiable but not a logical consequence of the ontology. Thus the user is informed that the answer is Yes, under condition $g\#Europe(f)$.

A query $neg(win(d))$ is compiled into $negation(win(d),[],Cnst,t)$, this query results in constructing a tu-tree for $win(d)$, a t-tree for $win(e)$, and a tu-tree for $win(f)$. The latter steps are already described above. The (only) leaf of the tu-tree for $win(d)$ is (equivalent to) $neg(g\#Europe(f))$, and the (only) answer obtained for $negation(win(d),[],Cnst,t)$ is (equivalent to) $g\#Europe(f)$. The answer for $neg(win(d))$ given for the user is the same as that for $win(e)$ in the previous case.

A (compiled) query $win(c,[],Cnst,t)$ results in two answers (equivalent to) $g\#Europe(f)$ and $neg(g\#Finland(f))$. Their disjunction is found a logical consequence of the ontology. Hence the answer returned for a query $win(c)$ is Yes.

Tabulation. The operational semantics described in Section 2 may result in d-trees with infinite branches. Also constructing an infinite set of d-trees is possible (due to recursion through negation). We use tabulation of XSB Prolog to discover infinite trees. The way in which it prunes infinite branches is sound w.r.t. our operational semantics, as the resulting tree has the same set of success leaves.

Unfortunately, the native XSB tabulation cannot be used to discover that an infinite set of d-trees is being constructed. This is because the tree constructing predicate appears in the first argument of `->`/2. XSB refuses to tabulate such predicates, and tabulation is implemented using an extra argument of the compiled predicates. If this tabulation discovers an infinite computation then case 2a of the definition of the operational semantics (Section 2) is applicable.

For Datalog normal programs, tabulation of XSB Prolog guarantees finiteness of computation. As the Herbrand base is finite, each infinite branch of a tree and each infinite sequence of trees can be discovered and pruned. This is not the case for Datalog hybrid programs (i.e. hybrid programs over a finite Herbrand universe). The reason is that the set of constraints over a finite Herbrand universe is not finite. Hence tabulation is not able to discover some infinite branches of a d-tree (and some infinite sequences of d-trees). Some additional safeness conditions [6] imply that the constraints of the

leaves of a d-tree are ground. Then the tabulation approach described above results in finite computations only. Under these conditions our implementation is complete for non floundering Datalog hybrid programs. (For a given program and goal, floundering means selecting a non ground negative rule literal.)

Handling DL constraints. DL-reasoners normally implement satisfiability verification of a knowledge base as the main reasoning service. All other services are reduced to the problem of checking satisfiability of the knowledge base [2]. A commonly offered service is to check if an individual (a) belongs to some concept (C). This service is reduced to satisfiability by extending the knowledge base with the axiom $\{a:\neg C\}$. The query $C(a)$ is then a logical consequence of the knowledge base if its extension is unsatisfiable.

Disjunctive queries are usually not offered as an explicit service by DL-reasoners. However, a disjunctive query $C(a) \vee D(b)$ can be reduced to checking unsatisfiability of the knowledge base extended with $\{a:\neg C,\ b:\neg D\}$ [1]. General disjunctive DL queries cannot in a straight-forward manner be solved in this way. Most DL logics do not consider negated roles (properties) to be valid expressions. Hence, using the same approach for roles is not feasible. This is why our prototype only allows concept literals (not properties) as constraints in programs.

In the general case, it may be necessary to delay constraint checking until the last step of query answering. If several nested derivation trees have been constructed during rule reasoning, a nested constraint is produced. That is, the constraint possibly is a conjunction of negated constraints, which in turn are (possibly existentially quantified) conjunctions and so on. However, nested constraints can be normalized into a conjunctive normal form (CNF) of concept literals. That is, a conjunction where each conjunct is a disjunction of concept literals (non-nested).

A conjunctive DL query $C_1 \wedge \ldots \wedge C_n$ where the conjuncts are disjunctions of concept literals can be answered in the following manner [9]. Each conjunct can be solved as described above. If each conjunct is a logical consequence of the underlying knowledge base, then so is the original conjunctive query (and vice versa).

It is a design decision when the obtained constraints are checked for satisfiability. In principle, such check should be performed for each constructed constraint. This is however too expensive. (On the other hand, this prunes d-tree branches as early as possible.) Currently the check is performed at completion of the main t-tree, this means once per goal. Alternative strategies are being considered, for instance performing the check at completion of each d-tree.

4 Conclusion

This paper describes a way of implementing HD-rules, an approach of combining non monotonic rules of Logic Programming (LP) with monotonic first order theories of Description Logic (DL). The approach has been introduced in [6]. Its declarative semantics combines the well-founded semantics of LP with the standard first order semantics of DL. An operational semantics is provided. Its main advantage is that an existing DL reasoner and existing Prolog engine can be re-used; hence the effort to construct an imple-

mentation is low. Here we implement a somehow simplified version of that operational semantics. Hybrid rule programs are compiled into XSB Prolog. A run-time system executes the compiled programs and interfaces a DL reasoner. The interface itself is programmed in Java, using Jena (and indirectly DIG). The compiler is written in XSB Prolog. The prototype is under development, and available at http://www.ida.liu.se/hswrl/.

Acknowledgement. This research has been partially funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

References

1. F. Baader, H.-J. Bürckert, B. Hollunder, W. Nutt, and J. H. Siekmann. Concept logics. Technical Report RR-90-10, 1990.
2. F. Baader, D. Calvanese, and D. McGuiness(et.al.), editors. *The Description Logic Handbook*. Cambridge University Press, 2003.
3. DIG. WWW Page. URL: `http://dig.sourceforge.net/`. Accessed 7 February 2007.
4. W. Drabent. What is failure? An approach to constructive negation. *Acta Informatica*, 32(1):27–59, Feb. 1995.
5. W. Drabent, J. Henriksson, and J. Maluszynski. Hybrid reasoning with rules and constraints under well-founded semantics. In *Web Reasoning and Rule Systems, Proceedings RR 2007*, volume 4524 of *Lecture Notes in Computer Science*, pages 348–357. Springer-Verlag, 2007.
6. W. Drabent and J. Maluszynski. Well-founded semantics for hybrid rules. In *Web Reasoning and Rule Systems, Proceedings RR 2007*, volume 4524 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2007.
7. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In *Proc. of European Semantic Web Conference*, pages 273–287, 2006.
8. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Well-founded semantics for description logic programs in the semantic web. In *RuleML*, pages 81–97, 2004.
9. I. Horrocks and S. Tessaris. A conjunctive query language for description logic aboxes. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 399–404. AAAI Press / The MIT Press, 2000.
10. Jena Semantic Web Framework. WWW Page, 18 August 2005. Available at `http://jena.sourceforge.net/`. Accessed 7 February 2007.
11. Miguel Calejo. InterProlog - a Prolog-Java interface. WWW page, September 2006. Available at `http://www.declarativa.com/interprolog/`. Accessed 7 February 2007.
12. Pellet OWL Reasoner. WWW Page, 14 March 2006. Available at `http://www.mindswap.org/2003/pellet/index.shtml`.
13. R. Rosati. Dl+log: Tight integration of description logics and disjunctive datalog. In *KR*, pages 68–78, 2006.
14. XSB. WWW Page. URL: `http://xsb.sourceforge.net/`. Accessed 7 February 2007.

Using Prolog as the fundament for applications on the semantic web

Jan Wielemaker[1], Michiel Hildebrand[2], and Jacco van Ossenbruggen[2]

[1] Human Computer Studies,
University of Amsterdam,
The Netherlands,
wielemak@science.uva.nl
[2] CWI, Amsterdam, The Netherlands
firstname.lastname@cwi.nl

Abstract. This article describes the experiences developing a Semantic Web application entirely in Prolog. The application, a demonstrator that provides access to multiple art collections and linking these using cultural heritage vocabularies, has won the first price in the ISWC-06 contest on Semantic Web end-user applications. In this document we concentrate on the Prolog-based architecture, describing experiences and vital aspects of the design.

1 Introduction

Prolog has some attractive properties for Web and Semantic Web applications. Safety and automatic memory management as well as incremental compilation are essential to web-programming, (natural) language processing, simple reasoning, constraint programming and a natural representation of the Semantic Web triple model are features that contribute to the usability of Prolog for web-programming. Disadvantages are lack of ready-to-use resources for dealing with Web protocols and documents as well as the availability of skilled Prolog programmers in this field.

Within the E-culture research program[3] we were in the luxury position to have access to a good Prolog based starting point [13] and contributing researchers with Prolog affinity and experience. A small demonstrator was extended into a award-winning application [9] by a team of five programmers spread over three institutes.

SWI-Prolog's features for Web-programming are described in detail in [14]. This document describes practical experience using the framework in a larger project. We concentrate on design aspects to facilitate re-usability and independence between the various components of the software.

This document is organised as follows. First we introduce the E-culture demonstrator, briefly describing its functionality and software architecture. Then we describe the libraries enabling the design, concentrating on those that have

[3] http://e-culture.multimedian.nl/

been added during the project to enhance modularity and reuse. In Sect. 7 we give some practical tips for deployment of a large Prolog-based server on the Web. We conclude with problems, lessons learned, related work and plans.

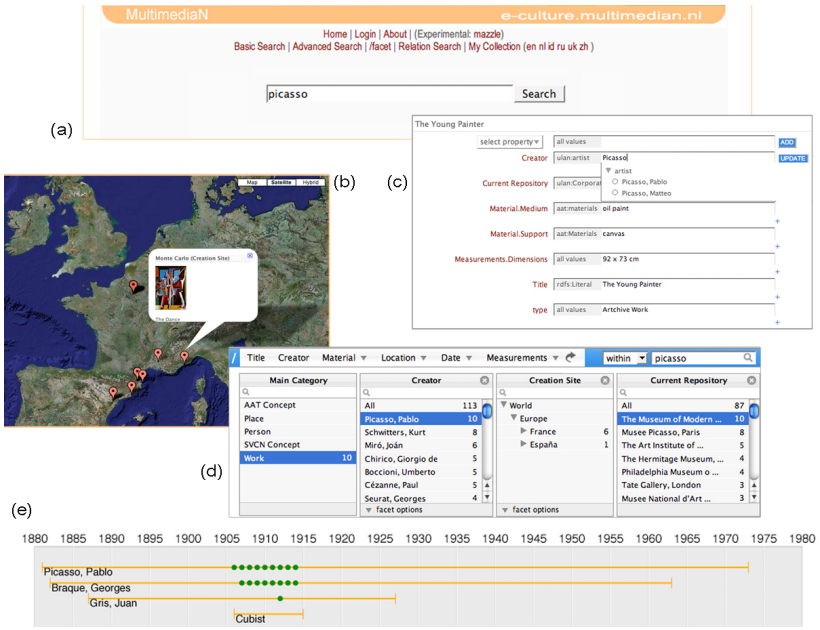


Fig. 1. Screendumps of the E-culture web-application. (a) simple text-based search interface, (b) geographical map visualisation, (c) resource annotation interface, (d) faceted navigation, (e) timeline visualisation.

2 Introducing the E-culture demonstrator

The aim of the E-culture demonstrator is to provide a common gateway to multiple museum collections and cultural heritage documents. Museums use different database models based on different vocabularies to represent their collection. Merging this into a single datamodel is complicated, labour intensive and leads to loss of information due to inadequacy of the common model as well as errors in the transformation process. We converted [11] both vocabularies and meta-data into RDF/OWL preserving the original structure. Only where literal strings were based on a known vocabulary, we restored the mapping to the vocabulary. After this lossless transformation process, the meta-data schema is mapped to the standard VRA schema[4] using RDFS subPropertyOf relations and cross-relations between vocabularies were restored or created. Our current RDF graph contains

[4] http://www.vraweb.org/

8.6 million triples describing over 100,000 art-objects from 4 different sources and 7 vocabularies.

The RDF graph is stored in memory [15] and made accessible from Prolog by means of the predicate **rdf**(*Subject, Predicate, Object*). The web-server of the demonstrator is realised by the SWI-Prolog multi-threaded HTTP server library[5]. In this web-server, a predicate serves one (typical) or more HTTP *locations*. The handler receives the parsed HTTP request as a Prolog data structure and writes a CGI document to the current output stream. This approach is comparable to Tomcat, where a class is defined to handle an HTTP location by writing a CGI document onto a stream.

Although any Prolog predicate that produces a valid CGI document can be used, the library html_write provides a DCG-based framework to write HTML and XHTML documents from the same specification. This library ensures proper nesting of tags and escapes for special characters. The library is described in [14].

The system contains two types of reusable modules. *Reasoning* modules on top of RDF provide RDFS (Schema) and limited OWL inferencing as well as more domain specific reasoning such as various graph-search and graph-abstraction predicates. *Presentation* modules define HTML DCG rules producing reusable components of the interface, such as presenting an image thumbnail or a widget that allows for selecting a term from a vocabulary using AJAX-based [7] interactivity.

Based on these reusable modules, different interfaces to the data are realised by different HTTP locations. Currently we have four interfaces. *Basic search* performs a graph-search from literals that match at least one word with the query to target objects (art-works) and clusters the results based on the RDF properties and class of the resource in the path from literal to target object. *Relation search* describes relations between arbitrary objects. */facet* provides a traditional facetted browser [5] and *Mazzle* merges basic search with facetted browsing while providing multiple points of focus, currently art-works, artists and geographical locations. Figure 1 shows some screenshots of the application, while the architecture is summarised in Fig. 2

3 Used technologies

It is an explicit aim of the project to use Open Standards where possible. This implies RDF/OWL for representing meta-data and vocabularies, a web-server (HTTP) using W3C standards for access. Machine-access is provided by means of the SPARQL[6] or SeRQL [2] RDF query language while human access uses browser standards.

Standard HTML has two limitations: lack of graphics and lack of interactivity. Initially these were resolved using SVG for non-interactive graphics and Java applets for interactivity. Eventually both have been replaced by HTML+CSS using AJAX for interactivity. HTML+CSS has limited graphical capability, but

[5] http://www.swi-prolog.org/packages/http.html
[6] http://www.w3.org/TR/rdf-sparql-query/

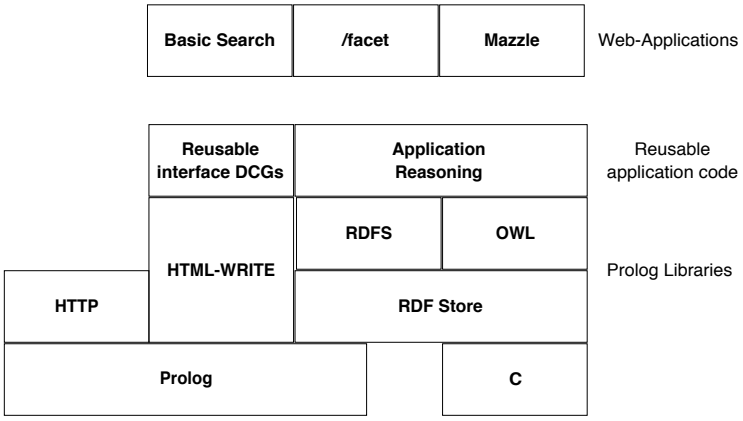


Fig. 2. Architectural components of the Prolog-based web-application

sufficient for our needs and they are much better supported by todays browsers. HTML+CSS with AJAX can deal with the interactivity we require, such as suggesting relevant vocabulary terms on each key-stroke in a text entry field. (Re)usable AJAX client scripts are widely available. Providing the required HTTP service that connects them to the data is easy.

4 Core Web libraries

In this section we describe the core libraries that enable the design. Some libraries have been described in other publications, in which case we keep the description concise.

4.1 The RDF library

The RDF library [15] is the core of SWI-Prolog's Semantic Web infrastructure. The key predicate is **rdf**(*Subject, Predicate, Object*), providing very natural access to the triple store. The predicate itself is defined in C. Because we know all clauses are ground unit clauses, resources are atoms and predicates are organised in a hierarchy using rdfs:subPropertyOf we can design an optimal representation minimising space and optimising access times. During the E-culture project we realised several enhancements to the core RDF library that are not described in previous publications and which we describe below.

Multi-threading support is enhanced by introducing *read-write locks* and *transactions*. During normal operation, multiple readers are allowed to work concurrently. Transactions are realised using **rdf_transaction**(*:Goal, +Context*). If a transaction is started, the thread waits until other transactions have finished. It then executes *Goal*, adding all write operations to an agenda. During this phase the database is not actually modified and other readers are allowed to proceed.

If *Goal* succeeds, the thread waits until all readers have completed and updates the database. If *Goal* fails or throws an exception the agenda is discarded and the failure or error is returned to the caller of **rdf_transaction/2**. Note that this behaviour is different from multi-threaded Prolog assert/retract.

- In multi-threaded (SWI-)Prolog, accessing a dynamic predicate for read or write demands synchronisation only for a short time. In particular, readers or writers with a choice-point allow other threads to operate on the same predicate. At the same time logical update semantics are realised. This is achieved using time-stamps and keeping erased clauses around until the predicate is sufficiently 'dirty' and there are no readers or writers.
- Multiple related modifications are bundled in a transaction. This is often desirable as many high-level (RDFS/OWL) changes involve multiple triples. Using transactions guarantees a consistent view of the database and avoids partial modifications.

RDF literals have been promoted to first class citizens in the database. Typed literals are supported using arbitrary Prolog terms as RDF object. Numbers (float, integer) are store in their native C representation, Unicode strings are stores as Prolog atom-handles and other Prolog terms are stored using the recorded-database access provided by SWI-Prolog through the foreign interface by means of PL_record(), PL_recorded() and PL_erase(). All literals are kept in an AVL-tree, where

$$\text{numeric-literals} < \text{string-literals} < \text{term-literals}$$

Numeric literals are sorted by value. String literals are sorted alphabetically, case insensitive and after removing diacritics. String literals that are equal after discarding case and diacritics are sorted on Unicode value. Other Prolog terms are sorted on Prolog standard order of terms. Sorted string literals are used for fast prefix search which is important for suggestions and disambiguation as-you-type with AJAX style interaction.

The literal search facilities are completed by means of *monitors*. Using **rdf_monitor**(*:Goal, +Events*) we register a predicate to be called at one or more given events. Monitors that trigger on literal creation and destruction are used to maintain a word-index for the literals as well as an index from stem to word and metaphone [8] key to word. Monitors are also used to achieve *persistency*. For persistency, each named graph is backed up by a file containing the state after initial load or last check-point and a file describing actions on the named graph, the *journal*.

4.2 Library HTML write

The HTML writer library uses Prolog DCGs in 'write' mode to translate a ground Herbrand term into a list of HTML tokens. The tokens are written to a Prolog stream using **print_html/2** to produce valid HTML. The Herbrand term can have embedded *term* sequences, which causes nested invocation of the DCG

referenced by *term*. We introduce the HTML library using an example from the OpenID[7] library. Note the in-line invocation of the rules **openid_title//0** and **hidden//2**. Details have been described in [14].

4.3 Session management

The core HTTP library defines a hook to expand the HTTP request. This hook is exploited by the session management library to realise cookie-based session management. The session library also defines **http_session_assert**(*+Term*), **http_session_retract**(*?Term*) and common assert/retract variations to realise storage of session specific data which can be queried using **http_session_data**(*?Term*).

Session-data is automatically retracted after session timeout. Start and end of a session is broadcasted (see Sect. 4.6), to enable additional processing by individual modules.

4.4 The HTTP dispatching code

The core HTTP library, described in [12], handles all requests through a single predicate. Normally this predicate is defined 'multifile' to split the source of the server over multiple files. This approach proved inadequate for a larger server with multiple developers for the following reasons:

- There is no way to distinguish between non-existence of an HTTP location and failure of the predicate due to a programming error. This is an omission in itself, but with a larger project and multiple developers it becomes more serious.
- There is no easy way to tell where the specific clause is that handles an HTTP location.
- As the order of clauses in a multi-file predicate that come from different files is ill defined, it is not easy to reliably redefine the service behind a given HTTP location. Redefinition is desirable for re-use as well as for experiments during development.

To overcome these limitations we introduced a new library http_dispatch that defines the directive **http_handler**(*Location, Predicate, Options*). The directive is handled by **term_expansion/2** to manage a multi-file predicate. This predicate in turn is used to build a Prolog term stored in a global variable that provides fast search for locations. Modifications to the multi-file predicate cause re-computation of the Prolog term on the next HTTP request. *Options* can be used to specify access rights as well as a priority that allows for overruling existing definitions. Typically, each location is handled by a dedicated predicate. Based on the handler definitions, we can easily distinguish failure from non-existence as well as find, edit and debug the predicate implementing an HTTP location.

[7] http://openid.net/

```
%%      openid_login_form(+ReturnTo, +Options)// is det.
%
%       Create the OpenID form.  This is exported as a separate DCG,
%       allowing applications to redefine /openid/login and reuse this
%       part of the page.

openid_login_form(ReturnTo, Options) -->
        { option(action(Action), Options, verify)
        },
        html(div(class('openid-login'),
                 [ \openid_title,
                   form([ name(login),
                          action(Action),
                          method('GET')
                        ],
                        [ \hidden('openid.return_to', ReturnTo),
                          div([ input([ class('openid-input'),
                                        name(openid_identifier),
                                        size(30)
                                      ]),
                                input([ type(submit),
                                        value('Verify!')
                                      ])
                              ])
                        ])
                 ])).

hidden(Name, Value) -->
        html(input([type(hidden), name(Name), value(Value)])).

openid_title -->
        html(div(class('openid-title'),
                 [ a(href('http://openid.net/'),
                     img([ src('file?name=openid_logo'), alt('OpenID') ])),
                   span('Login')
                 ])).
```

Fig. 3. HTML DCG presenting OpenID login page.

4.5 Setting management

Managing settings of the application is not typical for Web-servers, but the size of this project raised the need for central management of settings. Initial management was based on a file called `parms.pl` that defined **setting/1**, containing clauses like `setting(thumbnail_size(100,100)).` As the project grew we realised it was difficult for different developers to maintain different values for the settings without corrupting the central file under CVS revision control and this central file, holding information for many modules, seriously harmed modularity of the application and we introduced two new libraries. One for declaring, storing and asking setting values and one for querying and editing settings through the web-interface.

Declaration of a setting is achieved using the directive **setting**(*:Name, +Type, +Default, +Comment*). Settings are local to a module. Settings from other modules can be defined and requested using the standard ⟨*module*⟩:⟨*name*⟩ syntax instead of using a plain atom for the name. The interface includes **setting**(*:Name, -Value*), **set_setting**(*:Name, +Value*), **save_settings**(*+File*) and **load_settings**(*+File*). When settings are saved to file, only those that have a value not equal to their default are saved. Setting default declarations provide syntactical constructs to ask for environment variables and the value of other settings. Numerical settings can use arithmetic expressions and textual settings can use the + operator for concatenation.

Whenever a setting is modified the broadcast library described in Sect. 4.6 is informed. This allows modules to react on changes to settings immediately, also for settings that are only read during initialisation of the service.

The result provides distributed declaration of settings that no longer harms modularity. Proper typing and comments simplify reuse of settings over the application and an extensible web-interface manages the application settings.

4.6 The broadcasting service

The Prolog library broadcast was initially developed for the graphical subsystem XPCE to deal with application *events* and distributed information gathering. Its function can be compared to *hooks*, but central administration makes it easier to inspect broadcasted events and check who is listening to what events. The *hooks* are called *listeners* and are owned, where the owner is represented by an arbitrary ground term. When omitted, this is the module-name making the registration. We illustrate the functionality using a simple session. The atom `me` represents the owner. Details and source can be requested from the SWI-Prolog documentation server[8].

```
?- listen(me, hello(X), format('Hello ~w~n', [X])).
?- broadcast(hello(world)).
Hello world
?- unlisten(me).
```

[8] http://gollem.science.uva.nl/SWI-Prolog/pldoc/

```
?- broadcast(hello(world)).
```

Where **broadcast/1** runs a failure driven loop over all listeners, **broadcast_request/1** is non-deterministic and succeeds on any listener that succeeds.

The web-libraries use the broadcasting service for session and setting management.

5 SWI-Prolog enabling features

Discussed with more detail in [14], we will briefly summarise the requirements on Prolog that enable its use as Semantic Web application platform.

- Scalability requires for a multi-threaded Prolog engine. Next to exploiting multi-CPU hardware efficiently, it also avoids slow queries from making the server inaccessible.
- Using unlimited-length Unicode atoms and atom garbage collection allows for uniform and simple representation of arbitrary text for web-applications.
- The system requires support for incremental compilation, so code can be modified and the server can be updated and tested without restart or loosing sessions. SWI-Prolog offers **make/0**, which reloads all modified source-files comfortably. Currently, temporal inconsistencies in the running program during reload can cause errors in services that run concurrently. We plan to enhance this using read-write locks that synchronise program update with the HTTP worker threads. Lacking these locks is generally no problem for local development or non-critical public services.

6 The role of RDF query languages

Most Semantic Web applications are modelled after relational database applications, where the application logic accesses the database through SQL. We see a number of Semantic Web equivalents to SQL, such as SeRQL [2] and the W3C recommendation SPARQL[9]. Both allow for specifying a graph expression consisting of a number of obligatory and optional edges and nodes extended with conditions on literal values, SeRQL matches the graph expression on the transitive closure using the semantics of RDFS. The SPARQL standard does not specify whether or not entailment reasoning is performed by the database engine. We implemented SeRQL and SPARQL support on top of the SWI-Prolog Semantic Web library using the HTTP infrastructure defined in this document to make the server accessible for both humans and machines.

The E-culture application, however, does not use SeRQL or SPARQL. Instead, queries by the application logic are expressed as Prolog goals on the raw RDF database and/or RDFS/OWL reasoning modules. At places where the order of executing conjunctions is critical and cannot easily be predicted by the

[9] http://www.w3.org/TR/rdf-sparql-query/

application programmer, we use the query optimiser we developed for the SeRQL server [13], which rewrites a Prolog goal involving multiple calls to **rdf/3** and tests for optimal performance. Semantic Web query languages are not used in the application logic because

- Prolog itself already provides a completely transparent and easy to use API. As the application programmer uses Prolog anyway, Prolog syntax is a natural choice. Note that a classical approach for accessing relational databases from Prolog is by translating Prolog goals into SQL statements [6]. We see only a role using a query language for access by external applications and if query expressions are used to specialise the application for a specific environment and this specialisation is done outside the application itself.
- SPARQL lacks expressiveness to construct complex path expressions. For example, SPARQL does not support regular expressions in query paths, therefore, there exists no query that gets the root of a resource given a transitive property. Note that PSPARQL [3] is being developed to support exactly this.
- For our purpose we often need specific RDFS/OWL reasoning support in different parts of the demonstrator. Partial reasoning that fulfil our requirements is easily implemented and performs well. We believe efficient complete DL-reasoning over our large and generally inconsistent RDF store is not realistic.
- We have a need for dedicated graph search in which we guarantee quick termination by limiting the 'semantic distance' based on weighted relations.
- Current Semantic Web query languages support for literal search is generally limited to regular expression search and numerical conditions. We have need for searching for keywords that can appear inside literals, possibly considering stemming. We also require fast prefix search for the suggestion interface, both on full literals and on keywords. Many applications solve this problem by populating a general text indexing engine such as Lucene[10] with the literals.

  Indexing integrated with the RDF store, however, greatly reduces memory requirements and access times, while simplifying maintenance when the RDF store is modified.

7 Deployment

Like Apache, Tomcat, etc., the Prolog based HTTP server can talk directly to a standard compliant browser. This setup, running the Prolog server interactively from a non-privileged port is normally used by the developers.

With some care, public deployment can also use the Prolog server directly. On a typical Unix system this requires the server to be started as root and make the required system calls available from Prolog to drop privileges after opening the server port. Typically this setup asks for a dedicated, possibly virtual, server machine. Due to practical considerations we opted for the option to use a public

[10] http://lucene.apache.org/

Apache server as reverse proxy. It also allows placing the Prolog server inside a firewall and realises a greater level of reliability because ill-formed requests are already blocked by the proxy server. The configuration file below makes the demo available from apache. Apache requires the standard modules `proxy` and `proxy_http` to be enabled The Prolog server listens to port 3020.

```
ProxyPass        /demo/ http://mn9c.mydomain.org:3020/demo/
ProxyPassReverse /demo/ http://mn9c.mydomain.org:3020/demo/
```

The Prolog server is started from a Unix boot script. Maintenance of the E-culture demo such as re-loading modified Prolog source files using **make/0** is realised by means of HTTP commands. The SWI-Prolog documentation server[11] is realised with a similar setup, but the Prolog server runs interactively in a terminal inside a VNC server session using an unprivileged user that is started from a Unix boot script. This setup allows easy monitoring and modifications by contacting the VNC virtual desktop.

8 Metrics

Our current RDF store contains 8.6 million triples while we plan to deal with 150 million triples on a server with 8 CPUs and 32GB main memory within 2 years. The application specific code is about 35,000 lines. The SeRQL/SPARQL infrastructure counts 18,000 lines. Finally, the SWI-Prolog HTTP library is 5,100 lines and the Semantic Web database 7,300 lines Prolog and 11,000 lines of C.

Time to load all 8.6 million triples from RDF/XML and Turtle source is 350 seconds. Time to restore from the file-based persistent database is 40 seconds. Timings are measured on an Intel core duo X6800@2.93Ghz using the 64-bit version of SWI-Prolog 5.6.34 under SuSE Linux 10.2. Initial load and restore are currently not multi-threaded.

Process' data size is 1.8GB (64-bit mode). Resources are represented as atoms. We counted 3,4 million atoms, 0.6 million for the literal index, 2.8 million for resources and literals and only 18,000 for the program.

The 8.6 million triples contain 1.9 million literals. The token and stem indices are built in 90 seconds and require 200MB memory. The token index contains 1.0 million words and numbers. The stem index has 380,000 stems.

We acquired some statistics on public server. During 3 days of operation using 8 worker threads on 2 CPUs it used 12,000 seconds CPU time, an average of 2.5% of the system capacity. Table 1 shows how calls to **rdf/3** are distributed over the possible instantiation patterns.

9 Problems experienced

Our server uses a large amount of not very well established technology. There is not much established technology in the Semantic Web world, making this unavoidable in that part of the application. For serving general web-pages however

[11] http://gollem.science.uva.nl/SWI-Prolog/pldoc/

Indexed			Calls
-	-	-	14,430
+	-	-	833,552
-	+	-	3,600
+	+	-	216,792,146
-	-	+	2,252,522
-	+	+	38,739,699
+	+	+	2,337,826

Table 1. Indexing pattern on rdf(Subject,Predicate,Object) calls after 3 days of operation.

there are many alternatives such as Tomcat servlets, jsp, php, asp, etc. Doing it all in Prolog greatly simplifies and enhances the performance in the interaction between the RDF store and the general web-page generation. It also greatly simplifies deployment. An installed version of SWI-Prolog and the hierarchy of Prolog source files are the only dependencies.

Upgrading a platform that had only be tested on small scale applications developed by one programmer to a large demanding application with multiple developers proved to be a challenge that requested the concurrent development of modules to deal with dispatching, session management and setting management. We also had to establish the best practices to use the infrastructure, notably to reach at proper re-usability of interface components. Affinity with Prolog programming in the whole team was necessary to make this work. We hope the matured Prolog libraries for web-programming with a planned Open Source release of the demonstrator provides a platform for other teams.

There were two main sources of bugs in the platform. One was still incomplete or false processing in both the HTML/HTTP infrastructure and the Semantic Web libraries. The other source of problems was found in the low-level RDF store, notably locking for thread-safety and memory management issues in the C-code.

10 Lessons learned

We started this project based on the SeRQL server running on top of the SWI-Prolog Semantic Web and HTTP libraries [13]. This system was fairly simple and small, handling about 50 HTTP locations that had largely be defined by the OpenRDF [2] project. It was developed by a single programmer. The E-culture project has a larger development team, is aiming at a demanding and stable server platform while the best way to support end-users based on Semantic Web data is explored using multiple prototype web interfaces.

It quickly became apparent that this required infrastructure and best-practice guidelines on how web-applications needed to be written for optimal re-usability and modularity.

– The http_dispatch library greatly enhanced the ability to find and debug code handling an HTTP location.
– The setting management library realises distributed management of application settings.
– Instead of mixing application logic, general HTML primitives and the specific code to handle a set of HTTP locations in a single file we started a libraries with HTML primitives, general primitives based on the Semantic Web libraries and more high-level application logic.

Note that the design as a web application makes it easy to deploy multiple user-interfaces concurrently on the same server from different HTTP locations. Based on a stable low-level RDF and HTML output routines, experimental code and (semi-) production code live together on the same server.

New explorations are not handled using a branch in the revision control system, but using a copy of the code running on another HTTP location. Not using CVS branches simplifies refacturing needed to deal with evolving new infrastructure such as the introduction of the dispatch, setting and session management libraries.

The HTML write library based on DCG with inline calling of other rules using the \-syntax proves to work well. It can generate both traditional HTML and XHTML from the same Prolog source and allows for easy reuse of common components. An open issue is the content of the HTML head, notably required references to CSS and Javascript files. We must consider a syntax where DCG components can specify required CSS and Javascript which is moved to the head in an extra rewriting step.

Initially interactivity and graphics was provided by means of Java applets running SeRQL queries on the server. Modifications required changing and recompiling the applet code and quite commonly restarting the browser. Later we moved the application logic from the applet to the Prolog server, only keeping the interface behaviour in the applet. With stable applets, we can now change the application logic on the server and deploy the changes using a simple **make/0** at the server.

In early development all interaction was handled server-side, which required a new HTTP request and an update of the entire page for each action. A more responsive solution is available with client-side programming in Javascript. Simple interactions for which all data is already available on the client side can be solved completely client side with Dynamic HTML, an example is the thumbnail browser in /facet.

If the interaction requires additional data, the XMLHttpRequest [7] allows this to be requested from to the server asynchronously. The server response, typically in XML or JSON, is then processed on the client side where it updates the HTML through the Document Object Model (DOM). The combination of these technologies, also known as AJAX, allows for rich interaction strategies while reducing the server workload.

Various interface widgets, such as trees and tabbed views, are publicly available in several JavaScript libraries. Furthermore, services for geographical map-

ping, timeline and calendar visualisations are easily integrated and updated with AJAX technology.

11 Future plans

Scalability will be tested against two axis. By incorporating more collections we plan to scale to 150 million RDF triples. As the system becomes more widely knows and serves a larger set of collections more user-friendly we anticipate higher loads. It is planned to test scalability on an 8 CPU system with 32 GB main memory.

As the connectivity between vocabularies grows, the graph-based algorithms require more selective exploration of the graph and different abstraction mechanisms to provide sufficiently simple abstractions to satisfy the user.

We also foresee that a larger part of reasoning in the system will be specified in standard (Semantic Web) languages. Notably OWL descriptions can be used to specify target objects and rules (SWRL) can be be used to express simple reasoning and mappings that cannot be expressed using subPropertyOf or owl:sameAs. Such expressions can be translated into Prolog programs and optimised before execution.

We plan to rewrite parts of the web-interface and base it on the Yahoo UI library[12]. Replacing our widgets by professional (web-)widgets enhances the look-and-feel and releases the project from browser compatibility issues. Data interchange with the server will be based on JSON[13].

12 Related work

As far as we know, there are no Prolog systems offering comprehensive support for web programming concentrating on the Semantic Web. Many Prolog systems offer some form of support for the HTTP protocol. The most widely known example is the PiLLoW library [4] developed by the Ciao Prolog team and available for at least Ciao, SWI-Prolog, SICSTus Prolog and YAP. In [14] we compare PiLLoW and the SWI-Prolog infrastructure for handling HTML documents. ProWeb [1] is an ALP-Prolog library aimed at embedded HTTP servers for controlling appliances. Its notion of *Request Processing Modules* (RPM) is probably comparable to our http dispatch library. Lack of details on RPM make an actual comparison impossible. WebLS by Amzi! [10] appears specialised for question-answering type of applications.

13 Conclusions

We presented the SWI-Prolog (Semantic) web application platform with the E-culture demo server. The platform combines an RDF in-core database that is

[12] http://developer.yahoo.com/yui/
[13] http://www.json.org/

seamlessly connected to Prolog with an HTTP server infrastructure, The award-winning web-application, developed by five researchers proves the applicability of Prolog for Semantic Web applications. All described infrastructure is available as Open Source under the LGPL license. The source of the application as a whole will be made available later during the project.

Acknowledgements

This research was supported by the MultimediaN project funded through the BSIK programme of the Dutch Government.

References

1. Manfred Bathelt, Ulrich Gall, Bernd Hindel, and Christian Kurzke. Accessing embedded systems via www: the proweb toolset. In *Selected papers from the sixth international conference on World Wide Web*, pages 1065–1073, Essex, UK, 1997. Elsevier Science Publishers Ltd.
2. Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: An architecture for storing and querying rdf and rdf schema. In *Proc. First International Semantic Web Conference ISWC 2002, Sardinia, Italy*, volume 2342 of *LNCS*, pages 54–68. Springer-Verlag, 2002.
3. cois Baget Jérôme Euzenat Faisal Alkhateeb, Jean-Fran˙RDF with regular expressions. Technical Report RR-6191, INRIA Rhône-Alpes, May 22 2007.
4. Daniel Cabeza Gras and Manuel V. Hermenegildo. Distributed WWW programming using (ciao-)prolog and the piLLoW library. *TPLP*, 1(3):251–282, 2001.
5. Michiel Hildebrand, Jacco van Ossenbruggen, and Lynda Hardman. /facet: A Browser for Heterogeneous Semantic Web Repositories. In *The Semantic Web - ISWC 2006*, pages 272–285, November 2006.
6. Matthias Jarke, Jim Clifford, and Yannis Vassiliou. An optimizing prolog front-end to a relational query system. *SIGMOD Rec.*, 14(2):296–306, 1984.
7. Linda Dailey Paulson. Building Rich Web Applications with Ajax. *IEEE Computer*, 38(10):14–17, 2005.
8. Lawrence Philips. The double metaphone search algorithm. *C/C++ Users J.*, 18(6):38–43, 2000.
9. Guus Schreiber, Alia Amin, Mark van Assem, Viktor de Boer, Lynda Hardman, Michiel Hildebrand, Laura Hollink, Zhisheng Huang, Janneke van Kersen, Marco de Niet, Borys Omelayenko, Jacco van Ossenbruggen, Ronny Siebes, Jos Taekema, Jan Wielemaker, and Bob J. Wielinga. Multimedian e-culture demonstrator. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 951–958. Springer, 2006.
10. Arvindra Sehmi and Mary Kroening. Webls: A custom prolog rule engine for providing web-based tech support. Technical report, Amzi! inc.
11. Mark van Assem, Maarten R. Menken, Guus Schreiber, Jan Wielemaker, and Bob J. Wielinga. A method for converting thesauri to rdf/owl. In *International Semantic Web Conference*, pages 17–31, 2004.
12. J. Wielemaker.

13. Jan Wielemaker. An optimised semantic web query language implementation in prolog. In Maurizio Baggrielli and Gopal Gupta, editors, *ICLP 2005*, pages 128–142, Berlin, Germany, October 2005. Springer Verlag. LNCS 3668.
14. Jan Wielemaker, Zhisheng Huang, and Lourens van der Mey. SWI-Prolog and the Web. Paper submitted to tplp, HCS, University of Amsterdam, 2006.
15. Jan Wielemaker, Guus Schreiber, and Bob Wielinga. Prolog-based infrastructure for RDF: performance and scalability. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *The Semantic Web - Proceedings ISWC'03, Sanibel Island, Florida*, pages 644–658, Berlin, Germany, october 2003. Springer Verlag. LNCS 2870.

Author Index