

Description Logics Programs: An Evaluation and Extended Translation

Raphael Volz,¹ Boris Motik,² Ian Horrocks,³ and Benjamin Groszof⁴

¹ Institute AIFB, University of Karlsruhe, Germany,
volz@fzi.de

² FZI, University of Karlsruhe, Germany
motik@fzi.de

³ Department of Computer Science, University of Manchester, UK
horrocks@cs.man.ac.uk

⁴ MIT Sloan School of Management, Cambridge, MA, USA
bgroszof@mit.edu

Abstract. Description logic is expected to play a key role in ontology modeling for the Semantic Web. However, logic programming paradigm has been researched extensively in the past and has a large user community, so interoperability between these two formalisms is a desirable and an important issue. In this paper we extend our existing work on translation of description logics ontologies to logic programs. In particular we handle additional expressive primitives, such as existentials, equality, functional properties and nominals. We show how to apply our approach for answering queries over DL ontologies. Finally, we empirically analyse the benefits of our approach in practise by comparing the performance of query answering using state-of-the-art description logics systems and using our translation in conjunction with available logic programming systems.

1 Introduction

Description Logic has become one of the most prominent knowledge representation formalisms, and with recent work on the Ontology Web Language (OWL) standard [17], it is likely to be of increasing importance in the Semantic Web context. On the other hand, logic programming is also an important knowledge representation paradigm, with several implemented systems (e.g., XSB [19] and variants of Prolog) and a large user community. Up until now, these two formalisms have been largely separate. Bridging this gap would help the Semantic Web to gain momentum by increasing both the size of the potential user base and the range of systems and applications able to exploit Semantic Web ontologies.

The Description Logic Programs (DLP) paradigm first developed in [7] addresses this issue, establishing a degree of interoperability by defining a meaning preserving translation between description logic ontologies and logic programs. In this paper we extend this work in several directions. Firstly, we extend the mapping, removing the asymmetry with respect to existential property restrictions, and thus allowing a commonly used subset of OWL to be fully captured in DLP. Secondly, we study some of the more expressive features of OWL, such as nominals and functional restrictions, and

show how these can also be translated into logic programs. Finally, we present a prototypical implementation that uses a logic programming system to reason with a DLP ontology, and we compare the performance of this prototype with that of a description logic reasoner when handling large DLP ontologies.

These extensions to the translation greatly increase the expressive power of DLP: using existentials enables the modelling of incomplete information, allowing a distinction to be made between stating that someone has a son (whose name is not known), that someone has a son named “John”, or that it is unknown if they have a son or not. Similarly, nominals allow concepts to be defined by referring to well-known instances. Finally, functional properties allow uniqueness conditions to be captured, which is particularly important for interfacing with relational databases, where primary keys are used to uniquely identify records.

We believe that our translation will also help the logic programming user community to understand description logic principles, by relating them to principles familiar in logic programming. Moreover, the translated ontology can be a module in a larger logic program, for example allowing combination with other rules and thereby avoiding some of the limitations of reasoning in description logics. In particular, this allows the limitations imposed by the tree-like structure of description logic definitions to be overcome.

The application of techniques developed in deductive databases also promises to be of benefit when dealing with very large knowledge bases. The main reason for this is that deductive databases focus on processing information in sets, rather than one-by-one as it is done in DL systems.

The rest of the paper is structured as follows. In section 2 we recapitulate the fundamentals of description logics and logic programming. In section 3 we present the operator μ for translating description logics into logic programs. In section 4 we discuss the issues involved in evaluating transformed logic programs. In section 5 we briefly present the prototype system implementing the translation. The results of an empirical performance analysis are presented in section 6.

2 Fundamentals

In this section we give a brief recapitulation of the formalisms we deal with, in particular, description logics and logic programming.

2.1 Description Logics

Among the multitude of available description logics, we chose to base our translation on the variant taken as the basis for the Web Ontology Language (OWL) [17]. OWL is expected to be a central standard for ontologies on the Semantic Web and corresponds to a very expressive description logic known as $\mathcal{SHOIN}(\mathbf{D}^+)$ [12, 11]. However, our results are not restricted to this description logic only.

Syntactically, description logic concepts are defined by means of simple concept expressions, where complex concept expressions can be built from simpler ones by combining them using various connectives, e.g. conjunction and union. Concept expressions

DL	FOL
\top	<i>true.</i>
$a : C$	$C(a)$
$\langle a, b \rangle : P$	$P(a, b)$
$C \sqsubseteq D$	$\forall x. C(x) \rightarrow D(x)$
$P^+ \sqsubseteq P$	$\forall x, y, z. (P(x, y) \wedge P(y, z)) \rightarrow P(x, z)$
$C_1 \sqcap \dots \sqcap C_n$	$C_1(x) \wedge \dots \wedge C_n(x)$
$P \equiv Q^-$	$\forall x, y. P(x, y) \iff Q(y, x)$
$\exists P.C$	$\exists y. (P(x, y) \wedge C(y))$
$\forall P.C$	$\forall y. (P(x, y) \rightarrow C(y))$
$\top \sqsubseteq \leq 1 P$	$\forall x, y, z. (P(x, y) \wedge P(x, z)) \rightarrow y = z$
$\{a_1, \dots, a_n\}$	$x = a_1 \vee \dots \vee x = a_n$
$C_1 \sqcup \dots \sqcup C_n$	$C_1(x) \vee \dots \vee C_n(x)$
$\neg C$	$\neg C(x)$
$\geq n P$	$\exists y_1, \dots, y_n. \bigwedge_{1 \leq i \leq n} (P(x, y_i))$ $\wedge \bigwedge_{1 \leq i < j \leq n} (y_i \neq y_j)$
$\leq (n-1) P$	$\forall y_1, \dots, y_n. (\bigwedge_{1 \leq i \leq n} (P(x, y_i)))$ $\rightarrow \bigvee_{1 \leq i < j \leq n} (y_i = y_j)$

Table 1. Syntax and Semantics of Description Logics

can be used to state various types of axioms, e.g. subset or equivalence relationships between concepts. Although the semantics of a description logic is usually given denotationally, in this paper we focus on the semantics established through correspondence with first order logic. In particular, the semantics of concept expressions is given by FOL formulas with one free variable, whereas the semantics of axioms is given by closed FOL formulas. The syntax and semantics of description logics is given in Table 1 and is based on [3].

OWL introduces several syntactic constructs that are convenience abbreviations of other $\mathcal{SHOIN}(\mathbf{D}^+)$ constructs. Without loss of generality we assume that all such shortcuts *HAVE* previously been eliminated according to the expansions in Table 2.

2.2 Logic Programming

Logic programming is a knowledge representation mechanism based on Horn clauses – FOL implications with only one literal in the implication head and conjunctions of literals in the body, with all variables universally quantified. Horn clauses are often extended with a closed-world negation and arithmetic predicates, thus resulting in a classical logic programming environment.

The elementary syntactic units of the language are terms, which can either be constant symbols such as names (so-called *atomic values*) and numbers (integer, float, etc.), variable symbols (usually written in capital letters) or compound terms of the form $f(t_1, \dots, t_n)$ where t_i are terms, representing an invocation of the function f with arguments t_1, \dots, t_n . Terms are used to form literals of the form $p(t_1, \dots, t_n)$, where p is a predicate symbol, meaning that p holds for t_1, \dots, t_n . Literals can be assembled in rules (also called *clauses* or implications) of the form $H :- B_1, \dots, B_n$, which intuitively mean that the head literal H is true if all body literals B_i are all true. For ex-

OWL Syntactic Shortcuts
disjointWith: $C_1 \sqsubseteq \neg C_2$
differentIndividualFrom: $\{i_1\} \sqsubseteq \neg\{i_2\}$
SymmetricProperty: $P \equiv P^-$
FunctionalProperty: $\top \sqsubseteq \leq 1 P$
InverseFunctionalProperty: $\top \sqsubseteq \leq 1 P^-$
domain: $\exists P. \top \sqsubseteq C$
range: $\top \sqsubseteq \forall P. C$

Table 2. Syntactic Shortcuts

ample, the following rule expresses the fact that uncles of a person are the brothers of a parent: $\text{uncleOf}(X, Y) :- \text{hasParent}(X, Z), \text{brotherOf}(Z, Y)$. Rules with empty bodies are called *facts* – they define things that are always true. Rules in the program may be *recursive*, meaning that some predicates occur in the head of some and in the body of some other rules. Further, rules typically must be safe, meaning that every variable in the rule must appear in at least one positive literal.

Queries are conjunctions of literals of the form Q_1, \dots, Q_n . For example, $\text{uncleOf}(X, Z)$ represents a query to retrieve all pairs where the first object is an uncle of the second one. Evaluating a query in a logic program means determining all variable substitutions for which the query conjuncts are all true.

3 Translating Description Logic Ontologies to Logic Programs

In this section we present the translation operator μ that translates a description logic ontology O into a logic program $\mu[O]$. The operator is applied to the ontology in a recursive fashion and is partial – some concept expressions can't be handled using Horn logic programs, in which case the result of the operator is undefined.

3.1 Existing Translations

In this subsection we briefly recapitulate the translations from our previous work we published in [7].

A-Box Assertions. Translation of A-Box concept membership assertions can be done only for assertions of the form $A(a)$, where A is an atomic concept name. All assertions of the form $C(a)$, where C is a concept expression, must be preprocessed by introducing a new atomic concept name I_i not appearing in the ontology, and replacing the assertion $C(a)$ by a new A-Box assertion $I_i(a)$ and a new T-Box axiom $I_i \sqsubseteq C$.

Now for the preprocessed ontology, each A-Box concept membership assertion of the form $A(a)$ is simply translated into the ground fact $A(a)$. Each A-Box role membership assertion of the form $R(a, b)$ is simply translated into the ground fact $R(a, b)$.

T-Box Concept Axioms. Operator μ is applied to each T-Box concept axiom and produces rules of the logic program as follows:

$$\frac{C \sqsubseteq D}{C \equiv D} \Rightarrow \frac{\mu[D](X) :- \mu[C](X)}{\mu[D](X) :- \mu[C](X) \quad \mu[C](X) :- \mu[D](X)}$$

Rules created in such a way contain concept expressions, so they can't be executed directly. In the rest of this section we present a series of rewrite rules that expand one concept expression in such a rule into rules containing simpler concept expressions. The process terminates when all concept expressions in all rules have been expanded. In our presentation H denotes some head predicate or concept expression and B denotes a conjunction of body predicates or concept expressions.

Atomic Concepts. Each atomic concept is represented by a unary predicate in the logic program. The translation of rules containing concept expressions consisting of atomic concepts may be done as follows:

$$\frac{\mu[A](X) :- B \quad \Rightarrow \quad A(X) :- B}{H :- \mu[A](X), B \quad \Rightarrow \quad H :- A(X), B}$$

T-Box Property Axioms. Axioms specifying property inclusion, equivalence, inverse properties and transitivity are translated into rules as follows:

$$\begin{array}{l} \frac{P \sqsubseteq R \quad \Rightarrow \quad R(X, Y) :- P(X, Y).}{P \equiv R \quad \Rightarrow \quad R(X, Y) :- P(X, Y).} \\ \frac{P(X, Y) :- R(X, Y).}{P \equiv R^{-} \quad \Rightarrow \quad R(X, Y) :- P(Y, X).} \\ \frac{P(X, Y) :- R(Y, X).}{P^{+} \sqsubseteq P \quad \Rightarrow \quad P(X, Z) :- P(X, Y), P(Y, Z).} \end{array}$$

Conjunction. Conjunctive concept descriptions are translated into rules as follows:

$$\frac{\mu[C \sqcap D](X) :- B \quad \Rightarrow \quad \begin{array}{l} \mu[C](X) :- B \\ \mu[D](X) :- B \end{array}}{H :- \mu[C \sqcap D](X), B \quad \Rightarrow \quad H :- \mu[C](X), \mu[D](X), B}$$

Disjunction. Disjunctive concept descriptions can only be used in the body of rules. Evaluating logic programs with disjunctions in the head is significantly more computationally complex and is therefore not in the scope of our translation:

$$\frac{\mu[C \sqcup D](X) :- B \quad \Rightarrow \quad \text{undefined}}{H :- \mu[C \sqcup D](X), B \quad \Rightarrow \quad \begin{array}{l} H :- \mu[C](X), B \\ H :- \mu[D](X), B \end{array}}$$

Universal Quantifiers. Concept descriptions containing universal quantifiers are translated into rules as follows (the symbol f must be a new, previously unused function symbol; variable Y is a new variable previously unused in the rule):

$$\frac{\mu[\forall R.C](X) :- B \quad \Rightarrow \quad \mu[H](Y) :- R(X, Y), B}{H :- \mu[\forall R.C](X), B \quad \Rightarrow \quad \text{Undefined}}$$

The first translation is obtained through following translations:

$$\begin{array}{l} \forall X. (\forall Y. (R(X, Y) \Rightarrow C(Y)) \Leftarrow B) \Leftrightarrow \\ \forall X \forall Y. (\neg R(X, Y) \vee C(Y) \Leftarrow B) \Leftrightarrow \\ \forall X \forall Y. (C(Y) \Leftarrow R(X, Y), B) \end{array}$$

Concept descriptions containing universal quantifiers in the body can't be translated into Horn logic programs, since it would require using negation: selecting all elements connected through R to C is equivalent to subtracting elements connected through R to objects which are not C from the set of all objects.

3.2 New Translations

In this subsection we present the translations for \top , \perp , existential quantifiers, equality, nominals and functional properties that are new in our work.

\top and \perp . Concept expressions containing \top and \perp are translated into rules as follows:

$$\frac{\mu[\top](X) :- B}{H :- \mu[\top](X), B} \Rightarrow \text{Delete the rule.}$$

$$\frac{\mu[\perp](X) :- B}{H :- \mu[\perp](X), B} \Rightarrow \text{Delete the rule.}$$

If B is empty, the second translation may result in an unsafe rule – variable X will occur in the rule head without occurring in the rule body. If that happens, the result of the operator is undefined. Further, the third translation results in a rule with an empty head. Such a rule is an integrity constraint – it specifies conditions which must not occur. Not all logic programming environments support integrity constraints, so in that case the result of μ is undefined.

Existential Quantifiers. Concept descriptions containing existential quantifiers are translated as follows (the symbol f must be a new, previously unused function symbol; variable Y is a new variable previously unused in the rule):

$$\frac{\mu[\exists R.C](X) :- B}{H :- \mu[\exists R.C](X), B} \Rightarrow \begin{array}{l} R(X, f(X)) :- B \\ \mu[C](f(X)) :- B \end{array}$$

The first translation is obtained by starting with the FOL encoding of the concept description, skolemising the existential quantifier in the head of the rule and splitting the conjunction in the head using the Lloyd-Topor transformation [14]:

$$\begin{aligned} \forall X.(\exists Y.(R(X, Y) \wedge C(Y)) \Leftarrow B) &\Leftrightarrow \\ \forall X.((R(X, f(X)) \wedge C(f(X))) \Leftarrow B) &\Leftrightarrow \\ \forall X.(R(X, f(X)) \Leftarrow B \wedge C(f(X)) \Leftarrow B) & \end{aligned}$$

The second translation was already presented in [7] and is obtained also by starting with the FOL encoding of the concept description and simply moving the existential quantifier from the body of the rule outside:

$$\begin{aligned} \forall X.(H \Leftarrow \exists Y.(R(X, Y) \wedge C(Y)) \wedge B) &\Leftrightarrow \\ \forall X \forall Y.(H \Leftarrow R(X, Y) \wedge C(Y) \wedge B) & \end{aligned}$$

Instance Equivalence. Many translations of OWL descriptions require specifying that two instances are equivalent. However, this predicate is typically not available in logic programming environments, so we must provide one. As presented in [9], the correct semantics can be captured by five axioms of the equivalence. Reflexivity, symmetry and transitivity ensure that the predicate express the algebraic properties of equivalence. The substitutivity axioms ensure the correct semantics of equivalence for function and predicate symbols.

$$\begin{aligned}
&= (x, x). \\
&= (x, y) :- = (y, x). \\
&= (x, z) :- = (x, y) \wedge = (y, z). \\
&= (f(x_1, \dots, x_n), f(y_1, \dots, y_n)) :- \{= (x_i, y_i) | 1 \leq i \leq n\}. \\
&Q(x_1, \dots, x_n) :- Q(y_1, \dots, y_n) \cup \{= (x_i, y_i) | 1 \leq i \leq n\}.
\end{aligned}$$

The reader may note that the axioms of substitutivity have to be instantiated for all predicates Q and functions f used in the rule base. Also, one may note that the reflexivity rule is unsafe, so one cannot execute it directly in a logic programming environment. We can provide a partial solution by explicitly instantiating a fact of the form $= (a, a)$ for all individuals in the program. However, this is only a partial solution – we’d have to do this for the set of all skolemised terms, but this set is unfortunately infinite. Further, many logic programming environments offer built-in predicates that can handle the reflexivity axiom efficiently.

Enumeration/Nominals. A naive translation of nominals $\{i_1, \dots, i_n\}$ would be to replace it with a new concept name N_i , along with membership assertions $N_i(i_j)$ for each nominal element. Such translation, however, is not correct, since the interpretation of nominals consists only of specified instances. Using Clark’s completion, this can be written like this:

$$\forall X.(N_i(X) \Leftrightarrow X = i_1 \vee \dots \vee X = i_n)$$

The naive translation axiomatises the Clark’s completion right to left. To specify completeness of the set, however, one would have to interpret the rule left to right, which would require disjunction in the rule head. However, nominals of only one element can be handled, since only one disjunction in the head remains:

$$\begin{array}{c}
\frac{\mu\{i\}(X) :- B \quad \Rightarrow \quad N_i(X) :- B}{H :- \mu\{i\}(X), B \quad \Rightarrow \quad H :- N_i(X), B} \\
\hline
N_i \text{ is defined as:} \\
\frac{N_i(i).}{= (X, i) :- N_i(X)}.
\end{array}$$

This translation can further be optimized for the special case of the hasValue construct of OWL, where we can avoid introducing new predicate by simply replacing a variable with the individual:

$$\frac{\mu[\exists R.\{i\}](X) :- B \quad \Rightarrow \quad R(X, i) :- B}{H :- \mu[\exists R.\{i\}](X), B \quad \Rightarrow \quad H :- R(X, i), B}$$

Cardinality Statements. As may be observed from the table Table 1, translation of cardinality constraints into first-order logic uses either disjunction in the head or the negation of the equivalence predicate. Horn logic systems don’t provide these features, so cardinalities can’t be handled directly.

However, the maximum cardinality restriction with the value of one in the head of the rule can be handled, since only one disjunct in the head of the rule remains. This

is especially important since the unique and unambiguous property syntactic shortcuts (cf. Table 2) are expanded into such cardinality restrictions. The translation can be done as follows:

$$\frac{\mu[\leq 1 R](X) :- B}{\Rightarrow} = (Y, Z) :- R(X, Y), R(X, Z), B$$

3.3 Expressive Features not Handled

Operator μ is undefined for concept expressions of the form $\mu[\neg A]$. The reason for this is that negation in description logics and logic programming has significantly different semantics. In description logics negation has so-called classical semantics, which is quite different from negation-as-failure, typically employed in logic programming. In a logic program something is false if it can't be proven to be true. E.g., unless the database contains `Composer(johann-sebastian)`, one can entail \neg `Composer(johann-sebastian)`. On the other hand, in classical logic, the lack of information that Johann Sebastian Bach is a composer doesn't allow concluding that he is not a composer. This is related to the closed-world assumption: a logic program assumes that it knows all relevant facts and everything else is assumed to be false. A description logic ontology, on the other hand, assumes it doesn't know all the facts, so it must be explicitly told which facts are false.

Further, our goal was to translate description logic ontology into Horn logic programs, which don't support disjunctions in the heads of the rules. Hence, such ontologies currently can't be handled using the DLP paradigm. This construct could be handled using extended logic programs. However, it is well known that evaluating such programs has high computational complexity, so we leave this for our future research.

3.4 Translation Example

Our translation can be illustrated through the ontology presented in Table 3, describing the life of Johann Sebastian Bach. The translation of the ontology into a logic program is shown in table 4. To save space we don't repeat A-Box assertions which are syntactically identical to the assertions in the ontology. Also, we show only the substitutivity axioms for one function and one predicate symbol.

4 Evaluating Translated Programs

In this section we discuss in more detail how program $\mu[O]$ can be used to answer queries about instances of ontology concepts.

4.1 Enumerating Concepts

Computing all instances of some concept C can be done by answering the query $C(X)$. However, there is an important distinction that one must be aware of: The translated program $\mu[O]$ may contain function symbols, denoting individuals whose existence is known, but whose name isn't known.

(T1) Woman \sqsubseteq Person	(T7) ancestorOf ⁺ \sqsubseteq ancestorOf
(T2) Man \sqsubseteq Person	(T8) marriedTo ⁻ \sqsubseteq marriedTo
(T3) Wife \sqsubseteq Woman \sqcap \exists marriedTo.Husband	(T9) \exists livesIn.{leipzig} \sqsubseteq LeipzigInhabitant
(T4) Husband \sqsubseteq Man \sqcap \exists marriedTo.Wife	(T10) Genius \sqcap Composer \sqsubseteq \forall hasComposed.Masterpiece
(T5) Father \equiv Man \sqcap \exists hasChild.Person	
(T6) hasChild \sqsubseteq ancestorOf	
(A1) \exists hasChild.Man(johann-ambrosius)	(A7) Man(johann-ambrosius)
(A2) Composer \sqcap Man(johann-sebastian)	(A8) livesIn(johann-sebastian, leipzig)
(A3) Person(wilhelm-friedemann)	(A9) Genius(johann-sebastian)
(A4) hasChild(johann-sebastian, wilhelm-friedemann)	(A10) hasComposed(johann-sebastian, matthaeus-passion)
(A5) Woman(anna-magdalena)	(A11) Woman(maria-barbara)
(A6) marriedTo(johann-sebastian, anna-magdalena)	(A12) marriedTo(johann-ambrosius, maria-barbara)

Table 3. Example OWL Ontology

(T1) Person(X) :- Woman(X).	hasChild(X , $f_1(X)$) :- Father(X).
(T2) Person(X) :- Man(X).	Person($f_1(X)$) :- Father(X).
(T3) Wife(X) :- Woman(X), marriedTo(X , Y), Husband(Y).	(T6) ancestorOf(X , Y) :- hasChild(X , Y).
(T4) Husband(X) :- Man(X), marriedTo(X , Y), Wife(Y).	(T7) ancestorOf(X , Y) :- ancestorOf(X , Y), ancestorOf(Y , Z).
(T5) Father(X) :- Man(X), hasChild(X , Y), Person(Y).	(T8) marriedTo(Y , X) :- marriedTo(X , Y).
Man(X) :- Father(X).	(T9) LeipzigInhabitant(X) :- livesIn(X , leipzig).
(S1) $= (f_1(X_1), f_1(X_2))$:- $= (X_1, X_2)$.	(T10) Masterpiece(Y) :- Genius(X), Composer(X), hasComposed(X , Y).
...	(S2) Person(X) :- Person(Y), $= (X, Y)$.
(A1) I_1 (johann-ambrosius). hasChild(X , $f_2(X)$) :- $I_1(X)$. Man($f_2(X)$) :- $I_1(X)$.	(A2) I_2 (johann-sebastian). Composer(X) :- $I_2(X)$. Man(X) :- $I_2(X)$.
...	

Table 4. Translation of Example into LP

Consider the ontology from Table 3. A query for instances of Man will return only johann-sebastian and johann-ambrosius. However, evaluating Man(X) in the program from the Table 4 will additionally return f_2 (johann-ambrosius). This result set denotes the fact that in each model of $\mu[O]$ there is some unique element dependent on johann-ambrosius whose name is unknown. Note, however, that the presence of that element is very important for correctly inferring Father(johann-ambrosius). To obtain the same answer to a concept enumeration query as from a description logic reasoner, one must filter skolem constants out from the query result. In Prolog this can be implemented using the extralogical built-in predicate compound, which succeeds if the argument is bound to a complex term and not to a constant. Our query for the instances of Man can then be written as Man(X), not(compound(X)). In rest of this paper we assume that this filtering is done externally to the logic program.

4.2 Recursive Definitions

The ontology from Table 3 contains recursive definitions (of concepts Husband and Wife). Generally, for such ontologies there are several possible interpretations. One typically considers only fixpoint interpretations – the interpretations where no new facts can be inferred by applying the ontology definitions. Still, there are several different fixpoint interpretations of Husband and Wife: (1) as empty sets, (2) as {johann-ambrosius} and {maria-barbara} respectively, (3) as {johann-sebastian} and {anna-magdalena} respectively or (4) as union of (2) and (3). Now (1) is the least fixpoint (i.e. the fixpoint interpretation containing the smallest amount of facts), whereas (4) is the greatest fixpoint (i.e. the fixpoint interpretation containing the largest amount of facts) exist. In [16] properties of fixpoint interpretations have been analysed and another, so-called descriptive semantics, has been proposed as well. This semantics is important since for some expressive description logics fixpoints may not exist.

Similar problems arise when the DL ontology is translated into a logic program. Cycles in ontology definitions will manifest themselves as recursive rules, which can also be interpreted under least or greatest fixpoint semantics. However, most, if not all logic programming environments evaluate queries only under least fixpoint. It is possible, however, to evaluate logic programs without function symbols alternatively under the greatest fixpoint semantics. The authors of this paper are not aware of any known way to compute the descriptive semantics interpretation using logic programming. On the other hand, tableau procedures compute precisely this semantics [4]. To summarize, the approach presented in this paper is most suitable if the least fixpoint interpretation is sufficient for the requirements at hand.

5 Prototype Implementation

We implemented a prototype system demonstrating the approach presented in this paper. The logic programming foundation is obtained by reusing the datalog engine of KAON [15] – an ontology management infrastructure developed by FZI and AIFB at the University of Karlsruhe. The flow diagram showing how an OWL ontology is processed using the system is shown in figure 1.

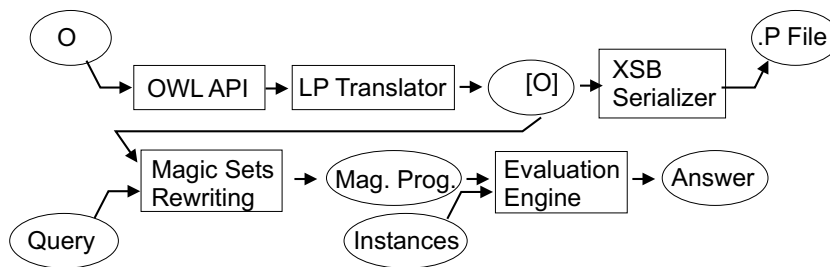


Fig. 1. Prototype Block Diagram

The input is given by the means of OWL API currently being developed at the University of Manchester and AIFB. An ontology O is then passed to the Logic Program

Translator implementing the translation from section 3, resulting in a logic program $\mu[O]$ represented using the KAON datalog engine API. This program can then be serialised as a Prolog file. Alternatively, one can supply a query and instance data and then evaluate the program.

Since the ultimate goal of KAON is to process larger quantities of data stored in relational databases, we have opted for the bottom-up query evaluation technique, which matches well with the way how databases evaluate queries. However, bottom-up evaluation has the drawback that the entire program is evaluated, including the part not relevant to the query. In order to improve this, we apply the magic sets transformation [1] to $\mu[O]$. Given a query, this transformation rewrites the program in a new program whose bottom-up evaluation will produce the same results with respect to the query. However, additional predicates introduced in the transformation ensure that only information relevant to the query is computed. In this respect the magic sets transformation simulates the top-down computation through bottom-up computation.

6 Performance Evaluation

In order to show the benefits of our approach, we have conducted a series of performance tests, the results of which we show in this section. We executed the tests with Racer version 1.7, XSB version 2.5 and our prototype described in the section 5. We didn't use the FaCT system [10] since it doesn't support A-Box assertions.

About Tools. Racer [8] was chosen as the state-of-the-art description logic reasoning system employing the tableau decision procedure as the inference mechanism. The implementation language of Racer is LISP. XSB [19] is a well-known logic programming system using SLG-WAM resolution and tabling [21] as the inference mechanism. The implementation language of XSB is C. The implementation language of KAON and of our prototype is Java.

Test Assumptions. Many description logic systems use caching extensively in order to speed up query processing. For example, once Racer computes the extension of some concept, it caches the results, so the next time the same query is issued, it is answered almost immediately. We decided not to take this into account since we wanted to measure the performance of query answering alone. It is quite obvious that, if the answer to the query is cached, query answering will be fast. Moreover, Racer doesn't perform incremental maintenance of query answers. We have observed that whenever the A-Box is changed, even if the change doesn't affect the result of the query, Racer forgets all cached information and answering the query takes the same amount of time as the first time.

Answering queries using XSB also took much longer the first time since XSB had to compile the logic program. However, we decided to ignore this time. Compilation of the program is not the same as caching the query result – each time the program is executed, the query is evaluated from scratch. Further, in XSB it is possible to assert or retract facts programmatically and this doesn't influence the speed of query answering.

Test Procedure. Each test is characterised by a certain ontology structure and a concept whose extension is to be read. The ontology structure has been generated for different input parameters, resulting in ontologies of different sizes. Obtained ontologies have then been loaded in each of the tools and the query has been executed. The average of five such invocations has been taken as the performance measure for each test. If executing the query took more than 15 minutes, the test was interrupted – results of such tests are denoted with MAX in the table.

Test Platform. We performed the tests on a standard PC with Pentium III processor running at 1.1 GHz, 380 MB of RAM running Windows XP operating system. Tests were written in Java and run using Sun's JDK version 1.4.1_01. Communication with Racer was done using JRacer library, whereas communication with XSB was performed through standard input and output.

Finally, before presenting the test results, we want to stress that we measured the performance of concrete tools. Although the algorithms used by all mentioned systems are certainly important, the overall performance of the system is influenced by many other factors as well, such the quality of the implementation or the language used to implement the system. It is virtually impossible to exclude these factors from the performance measurement.

6.1 Measurement Results

First we give an overview of the types of tests we conducted. In describing tests we use D to denote the depth of the concept tree, NS to denote the number of subconcepts at each level in the tree, NI to denote the number of instances per concept and P to denote the number of properties. The results of tests 1 to 6 are presented in Figure 2, whereas the results of the test 7 are presented in Figure 3.

Test 1. The goal was to see how the very basic tasks of traversing the concept hierarchy are handled. The ontology structure was a symmetric tree of directly classified concepts. The test was performed for $D = 3, 4, 5$; $NS = 5$; $NI = 10, 30$; $P = 0$. The ontology contained no properties and the query involved computing the extension of one of the first level concepts.

Test 2. The goal of this test was to see how ontologies with larger number of properties are handled. The ontology structure from Test 1 was extended with one property per concept, which was instantiated for every third instance of the concept pointing to the next instance. However, the properties were not mentioned in concept definitions (i.e. they were not relevant to the query at all). The test was performed for $D = 3, 4, 5$; $NS = 5$; $NI = 10$ and the query again was to compute the extension of one of the first level concepts.

Test 3. In the previous test we observed that the performance of Racer depended on the number of property instances, even if these are not mentioned in concept definitions. Hence, in this test we wanted to see whether smaller number of properties, but larger number of property instances will make a difference. The ontology structure from Test

1 was extended with a fixed number of properties. Each instance was connected with the previously generated instance through one property. The test was performed for $D = 3, 4, 5$; $NS = 5$; $NI = 10$; $P = 211$.

Test 4. The goal of this test was to test answering a simple conjunctive query. The ontology structure was identical to the one from Test 3, but the query was $c1 \sqcap \exists p0.c12$. The test for performed for $D = 3, 4$; $NS = 5$; $NI = 10$; $P = 3$.

Test 5. The goal of this test was to see how simple concept definitions are handled. The ontology structure consisted of a fixed number of properties. Each concept in the concept tree was defined using the following axiom: $c_i \sqcup \exists p_k.c_{i-1} \sqsubseteq c$ (where c_i denotes i -th child of concept c).

Test 6. The goal of this test was to show how larger quantities of information can be efficiently managed by storing them in the database and applying the transformation presented in this paper. We repeated Test 1 for $D = 6$; $NS = 5$; $NI = 10$, but stored instances in the database. We used the ODBC bridge of XSB to access the data from the database and we implemented a JDBC interface for KAON. The test was executed only with XSB and KAON, since Racer doesn't have a database interface.

Test 7. During execution of Test 3 we noticed that, although not relevant to the query result, the presence of property instances in the ontology has significant performance consequences in Racer. Hence, we repeated Test 3 with $D = 3$; $NS = 4$, while varying the number of instances per concept. We executed the test with ($P = 211$) and without properties ($P = 0$). We conducted the test with Racer only, since the goal was to show how presence of property instances significantly degrades the performance of reasoning in Racer. In XSB and KAON we observed no dependency of performance related to the number of property instances.

6.2 Discussion

From the results one can observe that the performance of Racer in all tests is significantly worse than the performance of XSB or KAON. We anticipate that this is mainly due to the fact that tableau reasoning procedure provides a proof or a refutation for one concept-instance pair, whereas SLG-WAM resolution or magic sets work with sets of instances. This is particularly true for answering conjunctive queries – SLG-WAM and magic sets evaluation take the full advantage of binding propagation, which limit the amount of computation to a minimum.

Further, one may observe that the performance of Racer deteriorates with the presence of property instances. For logic programs this makes absolutely no difference – the rules of the program don't reference predicates containing these property instances, so the amount of information stored in them isn't relevant. We don't have a plausible explanation for such behaviour in Racer.

Finally, one may observe that, as the amount of information grows, the information cannot be kept in main-memory, but must be stored in an external data storage. There we may observe that KAON still isn't as optimized as XSB. We presume that this stems

Test No.	Set	Concepts	Instances	Properties	Racer (s)	XSB (s)	KAON (s)
1	1	155	1550	0	12.18	0.16	0.18
	2	780	7800	0	95.17	0.47	0.83
	3	3905	39050	0	MAX	2.11	3.65
	4	155	4650	0	58.08	0.36	0.44
	5	780	23400	0	MAX	1.04	0.98
	6	3905	117150	0	MAX	MAX	5.62
2	1	155	1550	155	6.99	0.15	0.21
	2	780	7800	780	98.56	0.43	0.72
	3	3905	39050	3905	MAX	2.27	4.24
3	1	155	1550	211	5.39	0.21	0.39
	2	780	7800	211	MAX	0.56	0.88
	3	3905	39050	211	MAX	2.34	4.61
4	1	155	1550	3	34.30	0.11	0.82
	2	780	7800	3	MAX	0.64	3.07
5		155	1550	10	MAX	0.49	5.10
6		19530	195300	0	MAX	14.60	143.02

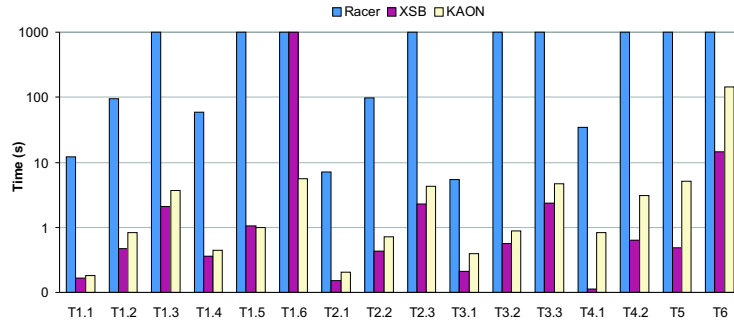


Fig. 2. Results of Tests 1 to 6

from the fact that SLG-WAM resolution uses tabling, which prevents it from computing some answers twice. KAON, on the other hand, uses bottom-up computation of magic programs. In our case most predicates are unary, so binding passing can occur only in a limited way. Therefore, the magic sets rewriting worsens the performance, rather than improving it. Obviously, this is the point where KAON needs improvement.

7 Related Work

Our work conceptually follows from the relationship between description logic and FOL ([3]) and multi-modal logic **K** ([20]). An axiomatisation of DAML+OIL, the precursor of OWL, was given by McGuinness and Fikes in [6]. Their axiomatisation language is the Knowledge Interchange Format (KIF). Therefore, their axiomatisation is not directly executable using logic programming systems, but a theorem prover is needed. Executing this axiomatisation using XSB was addressed in [22]. Obviously, some KIF rules can not be captured, such as Ax 105 and Ax 128. The current ver-

Test	Instances	No Prop. (s)	211 Prop. (s)
1	1550	10,48	4,63
2	3100	36,41	20,99
3	3225	48,94	30,24
4	3410	45,47	387,92
5	3875	71,06	624,48
6	4650	78,72	MAX

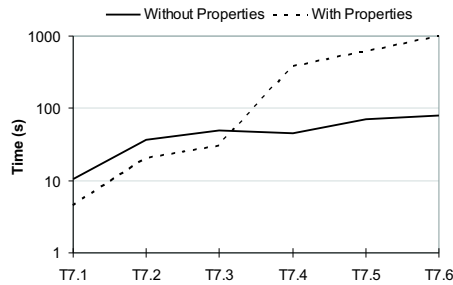


Fig. 3. Results of Test 7

sion is slightly outdated and incomplete, especially with respect to equivalence. Neither of these works addresses the slight semantical differences, such as those discussed in section 4.

Two other systems try to implement OWL using a rule-based formalism: The Euler Proof Mechanism [18] by Jos De Roo of Agfa and Tim Berners-Lee’s Closed World Machine (Cwm) [2] correspond very closely to our work and use the same syntactic format for rules. However, both approaches start from scratch and axiomatise even the elementary logic constructs. Their axiomatisation is not proven to be sound and/or complete either, e.g. the substitutivity of equality isn’t correctly captured. There is no formal characterization of the inference algorithms that these systems employ. In our approach, however, we rely on the well-known semantics and evaluation procedures of Horn logic and extensively reuse existing constructs for our purposes.

Finally, several systems, such as CARIN [13] or AL-log [5], attempted the integration of description logic with datalog rules. These systems don’t provide a translation of one formalism into another. Rather, they investigate which primitives from both formalisms can be successfully combined, resulting in a decidable system. Both systems rely on the tableau method for reasoning.

8 Conclusion

Motivated by the prospects of integrating description logic and logic programming, in this paper we presented an approach for translating a subset of OWL into logic programming. Based on this translation, we implemented a prototype system for handling OWL through logic programming. We conducted a series of tests comparing the performance of Racer, XSB and our own prototype system on ontologies of various sizes. From these tests we have observed that both XSB and our system perform on the order of magnitude better than Racer. This large difference in performance probably stems from the fact that description logics reasoners build a proof for one concept-instance pair at the time, whereas logic programming systems manage information in sets.

In future, we plan to investigate whether techniques of disjunctive deductive databases can be used to efficiently handle disjunction, negation and integrity constraints correctly. In particular, hyper-resolution and magic sets rewriting techniques seem promising in reducing query answering complexity in many practical cases.

References

1. C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 269–284, 1987.
2. Tim Berners-Lee. Cwm – Close World Machine. Internet: <http://www.w3.org/2000/10/swap/doc/cwm.html>, 2002.
3. Alexander Borgida. On the Relative Expressiveness of Description Logics and Predicate Logics. *Artificial Intelligence*, 82(1-2):353–367, 1996.
4. M. Buchheit, F. M. Donini, and A. Schaerf. Decidable Reasoning in Terminological Knowledge Representation Systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.
5. F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Al-log: integrating datalog and description logics. *Journal of Intelligent Information Systems*, 10:227–252, 1998.
6. R. Fikes and D. McGuinness. An axiomatic semantics for RDF, RDF Schema and DAML+OIL. Technical Report KSL-01-01, KSL, Stanford University, 2001.
7. B. Groszof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proceedings of WWW 2003*, Budapest, Hungary, May 2003.
8. V. Haarslev and R. Möller. RACER System Description. In *International Joint Conference on Automated Reasoning, IJCAR'2001*, Siena, Italy, June 2001.
9. S. Hoelldobler. *Foundations of Equational Logic Programming*, volume 353 of *LNAI*. Springer, 1987.
10. I. Horrocks. The FaCT System. In *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, pages 307–312. Springer-Verlag.
11. I. Horrocks and U. Sattler. Ontology Reasoning in the $\mathcal{SHOQ}(D)$ Description Logic. In *IJCAI-01*, pages 199–204, 2001.
12. I. Horrocks, U. Sattler, and S. Tobies. Practical Reasoning for Expressive Description Logics. In *Proc. 6th Int'l Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705, pages 161–180. Springer-Verlag, 1999.
13. A. Y. Levy and M.-C. Rousset. CARIN: A Representation Language Combining Horn Rules and Description Logics. In *European Conf. on Artificial Intelligence*, pages 323–327, 1996.
14. J. W. Lloyd. *Foundations of logic programming (second, extended edition)*. Springer series in symbolic computation. Springer-Verlag, New York, 1987.
15. B. Motik, A. Maedche, and R. Volz. A Conceptual Modeling Approach for Semantics-driven Enterprise Applications. In *Proc. 1st Int'l Conf. on Ontologies, Databases and Application of Semantics (ODBASE-2002)*, October 2002.
16. B. Nebel. Terminological Cycles: Semantics and Computational Properties. In J. F. Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, pages 331–361. Morgan Kaufmann Publishers, San Mateo (CA), USA, 1991.
17. P. F. Patel-Schneider, P. Hayes, I. Horrocks, and F. van Harmelen. Web Ontology Language (OWL) Abstract Syntax and Semantics. <http://www.w3.org/TR/owl-semantics/>, 2002.
18. Jos De Roo. Euler proof mechanism. Internet: <http://www.agfa.com/w3c/euler/>, 2002.
19. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In R. T. Snodgrass and M. Winslett, editors, *Proc. Intl. Conf. on Management of Data (SIGMOD 94)*, pages 442–453, 1994.
20. K. Schild. A correspondence theory for terminological logics: preliminary report. In *Proceedings of IJCAI-91, 12th International Joint Conference on Artificial Intelligence*, pages 466–471, Sidney, Australia, 1991.
21. Terrance Swift and David Scott Warren. Analysis of SLG-WAM Evaluation of Definite Programs. In *Symposium on Logic Programming*, pages 219–235, 1994.
22. Youyong Zou. Daml xsb interpretation. Version 0.3, Internet: <http://www.cs.umbc.edu/yzou1/daml/damlxsb.P.txt>, January 2001.