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]).

⋆ This is a slightly modified (a paragraph on p. 13 and the Acknowledgements) version of a paper presented at ICLP'07 Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS2007) and available at http://www.ceur-ws.org/Vol-287.

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,

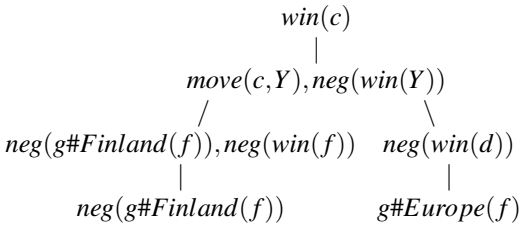
   a child is added to n with the label obtained from $q\theta$ by replacing s by $(B, Q)\theta$. If no such rule exists then n is marked as a failed node.
2. *s is negative*, i.e. of the form $neg(l)$. Two sub-cases are:
   (a) If l is non-ground, or recursion through negation has been discovered (see 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.

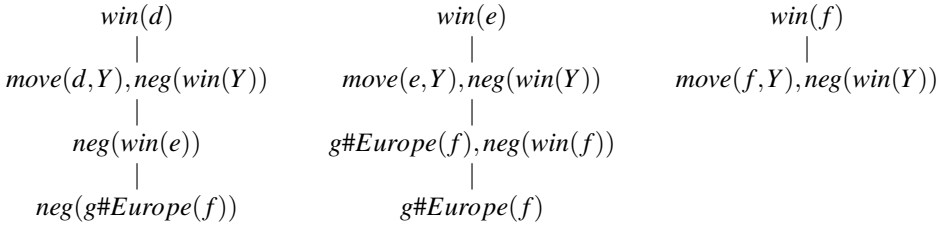
   (b) Otherwise l is ground; the step is completed after construction of a separate d-tree t for l. The kind of the separately constructed tree is different from the kind of the current tree, thus it is a tu-tree if the latter is a t-tree, and t-tree if the latter is a tu-tree. Let $C_1, \ldots, C_k$ be 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 $\backslash=/2$ (non-unifiability check) is employed to check disequality of nodes. (This could be done without built-ins, by replacing $E\backslash=G$ 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/.

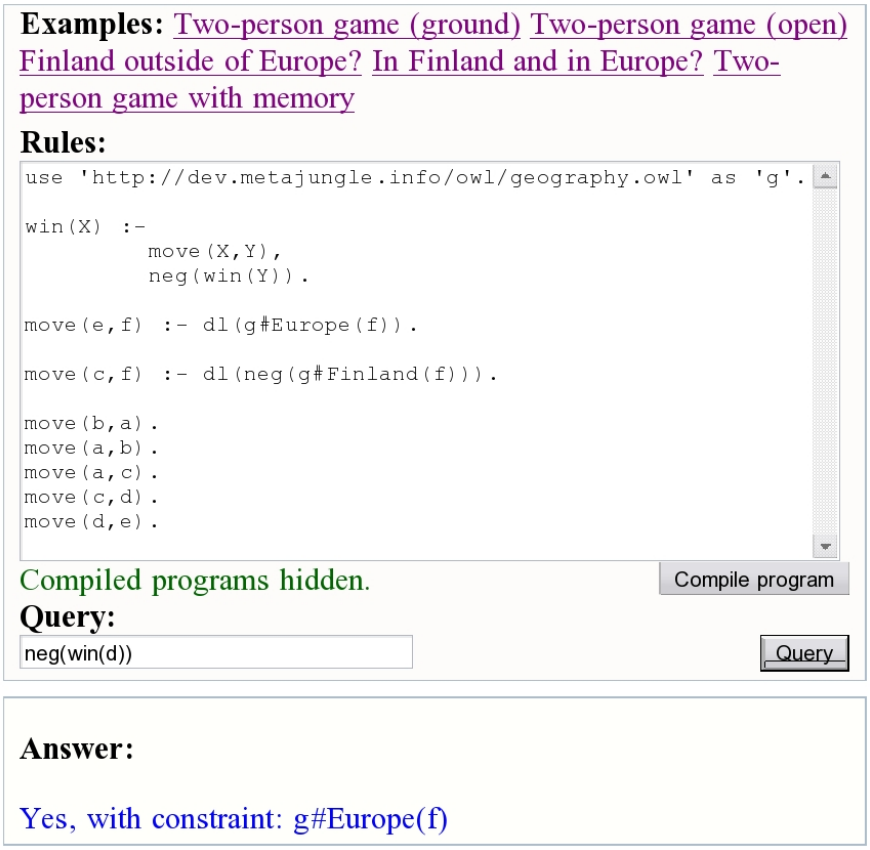


Figure 1. The web-interface of the hybrid reasoner answering a query with a constrained answer.

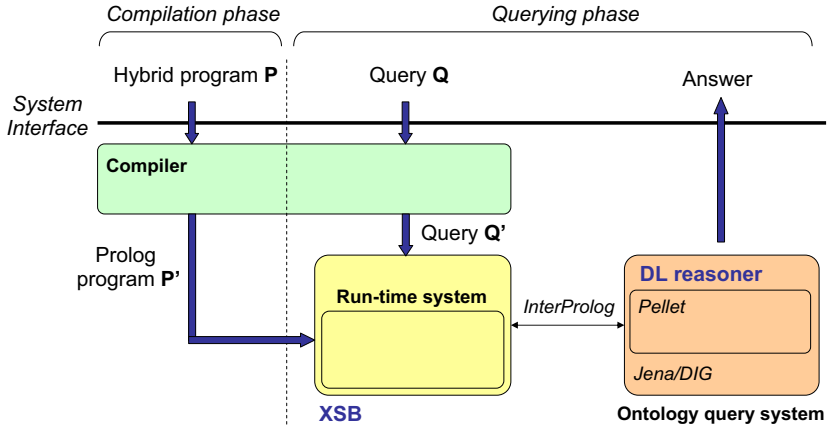


Figure 2. Prototype architecture overview.

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 occurrences of the same negated literal have to be treated differently (according to cases 2a or 2b of the description of the operational semantics in Section 2). Thus we tabulate $p(\bar{u})$ whenever a d-tree for $p(\bar{u})$ is built; the table is passed in an extra argument of the compiled predicates. Then infinite negation through recursion is discovered whenever predicate *negation*/4 is invoked with the first argument $p(\bar{u})$ found in the table being the second argument. Accordingly, a next d-tree for $p(\bar{u})$ is not constructed and case 2a is applied instead.

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 implementation 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`). We thank David S. Warren for a discussion about tabulation in XSB.

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.