A Tool for Knowledge Base Integration and Querying

Omar Elkhatib, Enrico Pontelli, Tran Cao Son
Department of Computer Science
New Mexico State University
{okhatib | epontell | tson}@cs.nmsu.edu

Abstract

We present a system (ASP−PROLOG) which provides a tight and well-defined integration of a multi-paradigm *logic programming system* (CIAO Prolog) and *Answer Set Programming* (ASP). The combined system allows the dynamic exchange of information between modules encoded using different logic programming paradigms. Each module might support a different form of reasoning (e.g., constraint solving, non-monotonic reasoning, functional-style computations), and modules are dynamic in nature, allowing a natural exchange of results and knowledge between them.

Introduction

In recent years, we have witnessed the development of several efforts to incorporate semantics and knowledge-based reasoning in question and answering (QA) systems (e.g., (Harabagiu 2001; Vicedo 2000; Pasca 2000)). This is necessary as queries, e.g., concerning actions, require the ability to deal with domain and commonsense knowledge, often not explicit in the text being processed. It is impractical to expect a single-paradigm framework to provide features that adequately support all forms of knowledge representation and reasonings. For example, commonsense knowledge can be nicely represented and manipulated using a non-monotonic logic programming language (e.g., *Answer Set Programming (ASP)*), but this framework is inadequate to deal with numeric reasoning or with reasoning based on processing large corpora. Thus, making an effective use of distinct sources of knowledge in QA tasks requires the ability to combine different models of knowledge representation and allow them to interact during the QA tasks.

Systems like CIAO Prolog (Hermenegildo *et al.* 1999) provide, through an articulated module system, the ability to develop communicating modules, that employ different flavors of logic programming. For example, in the case of CIAO, we can integrate traditional Prolog modules, modules based on *Constraint Logic Programming (CLP)* (over Finite Domains or over Reals), and modules with fuzzy logic programming capabilities. On the other hand, CIAO, as well as all the other general Prolog environments, do not provide any direct support for logic-based paradigms that naturally support commonsense and non-monotonic reasoning—such as Answer Set Programming (Niemela 1999). ASP is a computation paradigm in which logical theories serve as problem specifications and solutions are represented by *collections of minimal models (answer sets)*—a significant departure from the traditional goal-oriented and tuple-at-a-time view of computations of Prolog and CLP.

Most existing ASP inference engines are stand-alone systems, and have been extended to provide front-ends that are suitable to encode different types of knowledge. In spite of these extensions, there are aspects of reasoning that cannot be conveniently expressed in ASP:

- The development of an ASP program is mostly viewed as a monolithic batch process, and it does not support an interactive development of programs (as it is possible in the case of Prolog), where one can immediately explore the results of simply adding/removing rules.
- ASP systems offer very limited capabilities for reasoning on the *whole class* of answer sets associated to a program, e.g., to perform selection of models according to user-defined criteria. These activities are very important in many application domains—e.g., to express soft constraints and preference on models.
- ASP systems require grounding of programs before the computation of the answer sets—thus, making reasoning in presence of large domains (e.g., reasoning about numbers) impractical.
- ASP systems are independent systems; interaction with other languages can be performed only through low level and complex APIs.

We propose a system, called ASP−PROLOG, that represents a tight and semantically well-defined integration of ASP in Prolog. The language is developed using the module and class capabilities of CIAO Prolog. ASP−PROLOG allows programmers to assemble a variety of different modules to create a program; along with the traditional types of modules supported by CIAO Prolog, it allows the presence of an arbitrary number of *ASP modules*, each being a collection of ASP rules and facts. Each Prolog module can access any ASP module (using the traditional module qualification of Prolog), read its content, access its answer sets, and modify it (using the traditional `assert` and `retract` predicates).

Brief Semantic Foundations

Let us consider a language signature $\langle \mathcal{F}, \mathcal{V}, \Pi \rangle$, where

- $\mathcal{V}$ is a denumerable set of variables;
- $\mathcal{F}$ is a set of function symbols; in particular, $\mathcal{F} = \mathcal{F}_P \cup \mathcal{F}_A \cup \mathcal{F}_C$, where $\mathcal{F}_P$ are called *user* functions, $\mathcal{F}_A$ are called *ASP* functions, and $\mathcal{F}_C$ are called *interface* functions. We assume $\mathcal{F}_A \subseteq \mathcal{F}_P$ and $\mathcal{F}_A$ finite.
- Π is a set of predicate symbols, and $\Pi = \Pi_P \cup \Pi_A \cup \Pi_C$, where $\texttt{true}, \texttt{false} \in \Pi_P \cap \Pi_A$, Π_P are called *user-defined* predicates, Π_A are called *ASP-defined* predicates, and Π_C are called *Interface* predicates. In this work, we will focus on $\Pi_C = \{\texttt{assert}, \texttt{retract}, \texttt{models}\}$.
- $\mathcal{F}_A \cup \Pi_A \subseteq \mathcal{F}_C$.

We denote with $ar(f)$ the arity of f; we assume that $\forall f \in \mathcal{F}_A : ar(f) = 0$, and `assert`, `retract`, and `models` are all unary predicates.

The language adopted is multi-sorted, and it is based on the two sorts P (*Prolog*) and A (*ASP*). Each function (predicate) symbol f in $\mathcal{F}_P$ (Π_P) has sort $\mathsf{P}^{ar(f)} \rightarrow \mathsf{P}$ ($\mathsf{P}^{ar(f)}$). Each function (predicate) symbol f in $\mathcal{F}_A$ (Π_A) has sort $\mathsf{A}^{ar(f)} \rightarrow \mathsf{A}$ ($\mathsf{A}^{ar(f)}$). Also, the symbols in $\mathcal{F}_A$ and Π_A are of sort $\mathsf{A} \cup \mathsf{P}$. The sort A is used to identify terms and atoms that belong to ASP modules, while P is used for the construction of Prolog modules. We assume that terms and atoms are well-formed w.r.t. sorts. An atom built using symbols from Π_A and $\mathcal{F}_A \cup \mathcal{V}$ is called an *ASP-atom*; an atom built using symbols from $\mathcal{F}_P \cup \mathcal{V}$ and Π_P is called a *Prolog-atom*; an atom built using symbols from $\mathcal{F}_C \cup \mathcal{V}$ and Π_C is an *Interface-atom*.

Definition 1 *An* ASP-literal *is an ASP-atom A or its negation-as-failure (i.e., $not\ A$). An* ASP clause *has the form $A :\!- L_1, \ldots, L_n$ where A is a ground ASP-atom, and $L_1, \ldots, L_n$ are ground ASP-literals.*

Definition 2 *An* ASP constraint *is a conjunction $L_1 \wedge \ldots \wedge L_k$ of primitive ASP constraints of the type:*

- *an ASP-literal (A or $not\ A$); or*
- *a formula of the type $\alpha : L$ where α is a Prolog term and L is an ASP-literal.*

Definition 3 *An* Interface constraint *is a conjunction $L_1 \wedge \ldots \wedge L_k$ ($k \geq 0$) of primitive interface constraints of the type $\texttt{assert}(A :\!- B_1, \ldots, B_n)$ or $\texttt{retract}(A :\!- B_1, \ldots, B_n)$ or $\texttt{models}(t)$, where $A :\!- B_1, \ldots, B_n$ is an ASP clause, t is a P-term.*

Definition 4 *A $\mathbb{ASP}{-}\mathbb{PROLOG}$ rule is a formula:*

$$H :\!- C_1, C_2 \,[]\, B_1, \ldots, B_k$$

where H, C_1, C_2, and $B_1, \ldots, B_k$ are a Prolog-atom, an ASP-constraint, an Interface constraint, and Prolog-atoms, respectively. A static $\mathbb{ASP}{-}\mathbb{PROLOG}$ rule is a rule that does not contain `assert` *or* `retract`.

Definition 5 *An $\mathbb{ASP}{-}\mathbb{PROLOG}$ program*[1] *is a pair $\langle Pr, As \rangle$ where Pr is a set of $\mathbb{ASP}{-}\mathbb{PROLOG}$ rules and As is a set of ASP rules. A static program is a program $\langle Pr, As \rangle$ where all rules in Pr are static.*

[1] For simplicity, we focus on a single ASP module.

Operational Semantics

Let us denote with $\mathcal{H}_A$ ($\mathcal{H}_P$) the Herbrand universe built using the symbols in $\mathcal{F}_A$ ($\mathcal{F}_P$). The notation $\mathcal{H}$ will represent the complete Herbrand universe. We will also use the notation $\mathcal{B}_A$ (resp. $\mathcal{B}_P$, $\mathcal{B}$) to denote the Herbrand base obtained from the symbols of $\mathcal{F}_A \cup \Pi_A$ (resp. $\mathcal{F}_P \cup \Pi_P$, $\mathcal{F} \cup \Pi$). Let us start by focusing on static programs. The absence of `assert` and `retract` guarantees that the content of the As part of the program will remain unchanged throughout the execution.

Let $P = \langle Pr, As \rangle$ be a static $\mathbb{ASP}{-}\mathbb{PROLOG}$ program. The component As is a standard answer-set program (Niemela 1999); let us denote with

$$\mathcal{M}(As) = \{M \subseteq \mathcal{B}_A \mid M \textit{ is an answer set of } As\}$$

The semantics for P can be derived as a natural extension of the semantics of pure logic programming; the notion of model should simply be extended to accommodate for the meaning of ASP-constraints and interface constraints. The only additional element we require is a map used to *"name"* the models of the As part of the program. Let $\nu : \mathcal{M}(As) \rightarrow \mathcal{H}_P$ be an injective function, called the *model-naming function*. Then, a pair $I = \langle M, \nu \rangle$ is a model of the program if $M \subseteq \mathcal{B}_P$ and it satisfies all the Pr rules; I satisfies a ground primitive ASP-constraint and interface constraint if:

- A is an ASP-literal, then $\langle M, \nu \rangle \models A$ iff $\forall S \in \mathcal{M}(As)\ (S \models A)$
- A is an ASP-constraint of the form $t : B$, then $\langle M, \nu \rangle \models A$ iff $\exists S \in \mathcal{M}(As)\ (\nu(S){=}t \wedge S \models B)$
- A is a primitive interface constraint of the type $\texttt{models}(t)$, then $\langle M, \nu \rangle \models A$ iff $\exists S \in \mathcal{M}(As)\ (\nu(S){=}t)$

It is easy to extend these definitions to entailment of an arbitrary goal and to define the concept of satisfaction of clauses. Given a fixed model naming function ν, there is a unique minimal model $\langle M, \nu \rangle$ of P, according to the ordering $\sqsubseteq$ defined as: $\langle M_1, \nu \rangle \sqsubseteq \langle M_2, \nu \rangle$ iff $M_1 \subseteq M_2$.

Let us now proceed in extending the semantics structure when updates to the ASP theory are allowed through the `assert` and `retract` interface constraints. We will focus on an operational semantics. Let R be a *computation rule* (Lloyd 1987)—i.e., a function $R : \mathcal{B}^* \rightarrow \mathcal{B}$ which is used to select a subgoal.

Definition 6 *A* state *is a tuple $\langle G, \sigma, \tau, As \rangle$ where*

- *$G \in \mathcal{B}^*$ is called the* goal list
- *σ is a* substitution *(i.e., a function from $\mathcal{V}$ to $\mathcal{H}$)*
- *τ is a (partial) function $\tau : \mathcal{H}_P \rightarrow 2^{\mathcal{B}_A}$ called* model retrieval function
- *As is an ASP-program.*

Given a program $P = \langle Pr, As \rangle$, the initial state *is the tuple $\langle G_0, \epsilon, \tau_0, As \rangle$, where G_0 is the initial goal, ϵ is the empty substitution, and τ_0 is the function that is undefined for every input.*

Given a list $\bar{A}$, we denote with $[A_i/B]\bar{A}$ the list obtained by replacing the element A_i with the element B, and with $\bar{A} \backslash A_i$ the list obtained by removing A_i. Entailment is defined via a transition relation between states.

Definition 7 *Let $\langle G, \sigma, \tau, As\rangle$ be a state. The relation*

$$\langle G, \sigma, \tau, As\rangle \vdash_R \langle G', \sigma', \tau', As'\rangle$$

holds if $R(G) = A$ and:

- *if A is a P-atom, then there is a rule $H :- \bar{B} \in Pr$, such that $\theta = mgu(A, H)$, $\sigma' = \sigma \circ \theta$, $\tau = \tau'$, $As = As'$, and $G' = ([A/\bar{B}]G)\theta$.*
- *if A is an ASP-literal, then there is a ground substitution θ for A such that $\forall S \in \mathcal{M}(As)$ $(S \models A\theta)$, $G' = (G \setminus \{A\})\theta$, $\sigma' = \sigma \circ \theta$, $\tau' = \tau$, $As' = As$.*
- *if A is of the form $t : H$, then there is a grounding substitution θ for $t : H$ such that $\tau(t\theta)$ is defined, $\tau(t\theta) \in \mathcal{M}(As)$, $\tau(t\theta) \models H\theta$, $G' = (G \setminus \{A\})\theta$, $\sigma' = \sigma \circ \theta$, $\tau = \tau'$, $As = As'$.*
- *if A is of the form `models`(t), then there is a grounding substitution θ for t such that $\tau(t\theta)$ is defined, $\tau(t\theta) \in \mathcal{M}(As)$, $G' = (G \setminus \{A\})\theta$, $\sigma' = \sigma \circ \theta$, $\tau' = \tau$, and $As' = As$.*
- *if A is of the form `assert(r)`, then $G' = G \setminus \{A\}$, $\sigma' = \sigma$, $As' = As \cup \{r\}$, K is a set of terms from $\mathcal{H}_P$ (model names) such that $|K| = |\mathcal{M}(As')|$ and*
  - *for each $t \in K$ we have that $\tau(t)$ is undefined*
  - *$s_1, \ldots, s_r$ is an enumeration of K*
  - *$S_1, \ldots, S_r$ is an enumeration of $\mathcal{M}(As)$*
  - *$\tau' = \tau \circ \{s_1 \mapsto S_1, \ldots, s_r \mapsto S_r\}$*
- *if A is of the form `retract`(r), θ is a grounding substitution such that $r\theta \in As$, then $G' = (G \setminus \{A\})\theta$, $\sigma' = \sigma \circ \theta$, $As' = As \setminus \{r\theta\}$, K is a set of terms from $\mathcal{H}_P$ (model names) such that $|K| = |\mathcal{M}(As')|$*
  - *for each $t \in K$ we have that $\tau(t)$ is undefined*
  - *$s_1, \ldots, s_r$ is an enumeration of K*
  - *$S_1, \ldots, S_r$ is an enumeration of $\mathcal{M}(As)$*
  - *$\tau' = \tau \circ \{s_1 \mapsto S_1, \ldots, s_r \mapsto S_r\}$*

Given a program $P = \langle Pr, As\rangle$ and a goal G, we say that $P \models G\sigma$ iff $\langle G, \epsilon, \tau_0, As\rangle \vdash_R^* \langle \emptyset, \sigma, \tau, As'\rangle$.

The $\mathbb{ASP}-\mathbb{PROLOG}$ System

The $\mathbb{ASP}-\mathbb{PROLOG}$ system has been developed as an extension of the CIAO Prolog system (Hermenegildo *et al.* 1999). The choice of CIAO was fairly natural, being a flexible Prolog system, with a rich set of features aimed at facilitating the extension of the language (e.g., module system and object oriented capabilities). ASP modules are handled by *Smodels* (Niemela 1999).

Concrete Syntax

The abstract syntax presented in the previous section has been refined in the implementation of $\mathbb{ASP}-\mathbb{PROLOG}$ to better match the characteristics of Prolog. Each $\mathbb{ASP}-\mathbb{PROLOG}$ program is composed of a collection of *modules*. We recognize two types of modules: *Prolog* modules—which contain standard CIAO Prolog code—and *ASP* modules—each containing an ASP program. We will use an ASP program—called `plan.lp`—that solves planning problems in the block world domain, as a running example to illustrate the most important syntactically features of our system. For our purpose, it is enough to know that `plan.lp` consists of rules specifying the initial configuration, the goal configuration, and the effects of the actions (e.g., $move(a, b)$ will make a on b if nothing is on top of b, a) in this domain. The program has a parameter ($steps$) that determines the length of the plan. An execution of this program is obtained by issuing

```
lparse -c steps=5 plan.pl | smodels 0
```

which will return all the answer sets of `plan.pl`, each corresponding to a plan of length 5. We will now detail the syntax of $\mathbb{ASP}-\mathbb{PROLOG}$.

Module Interface Prolog modules are required to declare their intention to access any ASP modules; this is accomplished through the declarations

$$:- \texttt{use\_asp}(\textit{module\_name}, \textit{file\_name}, \textit{parameters})$$

where the *module_name* is the name used to address the ASP module, *file_name* is the file containing the ASP code, and *parameters* is a list of name/value parameters, to be passed to the ASP module.

Example 1 *A CIAO module might refer to the ASP* `plan` *module as follows:*

```
:- module(program1, [blocks_solve/0]).
:- use_asp(plan,'plan.lp',[steps(0)]).
```

The first line defines the CIAO module named `program1` *which exports the predicate* `blocks_solve`*. The second line declares that* `program1` *will access the ASP module* `plan` *with parameter $steps$ whose value is initiated with 0.*

Interface Constraints A number of predicates allows Prolog modules to query and manage ASP modules:

- `model/1`: in $\mathbb{ASP}-\mathbb{PROLOG}$, models of an ASP module can be retrieved using names; `model` binds its argument to the term representing the (internal) name of a model. This predicate has to be qualified with the ASP module on which it is meant to be applied. E.g., the goal `plan:model(Q)` will instantiate the variable Q with the (name of the) first model of `plan.pl`. The goal will fail if the program `plan.pl` does not have an answer set.
- `total_answer_sets/1`: the predicate is satisfied if the argument is the number of answer sets of the ASP module. E.g., `plan:total_answer_sets(X), X>0` succeeds if `plan.pl` has at least one answer set.
- `assert/1` and `retract/1`: the argument of these predicates is a list of ASP rules. The effect of `assert` is to add all the rules in the list to the ASP module, while `retract` will remove the rules from the ASP module. For example, if we are interested only in plans that do not move block a on the table during their execution, we can add an ASP-constraint that prevents the occurrence of the action $move(a, table)$. From Prolog, we can issue `plan:assert([:-move(a, table, T), time(T)])` which will add the constraint "`:-move(a, table, T), time(T).`" to `plan.pl`. We provide also non-backtrackable versions of these predicates (`assert_nb/1` and `retract_nb/1`), where the ASP updates are not undone upon backtracking.
- `change_parm/1`: most ASP inference engines allow the user to specify (typically as command-line arguments) various parameters that affect the ASP computation (e.g., initial value for constants). The predicate `change_parm` allows the user to read and modify the value of such pa-

rameters dynamically. The following Prolog fragment allows us to change the $steps$ parameter of `plan.pl`:

```
blocks_solve :-
  plan:total_answer_sets(X), X>0,
  chk_condition(1, X, Q),
  print_solution(Q, 0).
blocks_solve :-
  plan:change_parm([steps(V)]),
  V1 is V+1, plan:change_parm([steps(V1)]),
  blocks_solve.
```

Here, `chk_condition` will check whether a plan satisfies certain condition or not and `print_solution` will print the solution to the screen. The first call to `change_parm` will instantiate `V` to the current value of `steps`, while the second will modify the value of the constant.

- `compute/2`: this predicate has been introduced to specifically match another control feature provided by *Smodels*—it allows the presence of a compute statement, used to establish bounds on the number of models and to specify elements that have to be present in all the models. The `compute` predicate allows the Prolog module to dynamically affect these properties. For example, if we want to limit the maximum number of models to 3 in the ASP module `plan` and have all models contain `p`, then we can issue the goal `plan : compute(3,[p]).`
- `clause/2`: this predicate is used to allow a Prolog module to access the rules of an ASP module—in the same spirit as the `clause` predicate is employed in Prolog to access the Prolog rules present in the program. The two arguments represent respectively the head and the body of the rule.

Example 2 *Let us assume that the ASP module* `xyz` *contains the following rules for* `p`*:* $p(a) :- q(a), r(a)$ *and* $p(b) :- r(b)$*. Then the Prolog goal* `xyz:clause(p(X), Y)` *has two solutions:*

$$\{X \mapsto a, Y \mapsto (q(a), r(a))\} \quad \{X \mapsto b, Y \mapsto r(b)\}$$

Observe that, due to the fact that the syntax of *Smodels* is not ISO-compliant, certain *Smodels* constructs (e.g., cardinality and weight constraints) have a slightly different syntactic representation when used within Prolog modules. For example, if an ASP module (e.g., module `plan`) contains the rule $p :- 1\{r, s, t\}2.$ then the execution of the goal `plan:clause(p,X)` will produce the substitution

$$\{X \mapsto' \{\}'(1, (r, s, t), 2)\}.$$

ASP Constraints The syntax used to express ASP constraints is the same one described in the abstract syntax. E.g., if we would like to find plans that do not move block a to the $table$ (represented by the atom $move(a, table, t)$ where t is some number between 0 and $steps$), we can use the following rules:

```
chk_condition(Y, _, Q) :-
    plan:model(Y, Q), chk_cond(Q), !.
chk_condition(Y,X,Q):-Y=<X,Y1 is Y+1,
    chk_condition(Y1, X, Q).
chk_cond(Q) :-
    Q: move(a, table, _), !, fail.
chk_cond(_).
```

The next group of rules extract a plan from an answer set and display it on the screen:

```
print_solution(Q, T) :-
    Q:move(_,_, T), !, print_sol(Q, T),
    T1 is T+1, print_solution(Q, T1).
print_solution(_,_).
print_sol(Q, T) :- Q:move(X, Y, T),
  format(''~q on ~q time ~q'',
                [X,Y,T]), nl, fail.
print_sol(_,_).
```

System Implementation

The system is composed of two parts, a *preprocessor* and the actual CIAO Prolog system.

Preprocessing The input to the preprocessor is composed of *(i)* the main Prolog module (Pr); *(ii)* a collection of CIAO Prolog modules ($m_1, m_2, \ldots, m_n$); and *(iii)* a collection of ASP modules ($e_1, e_2, \ldots, e_m$). The output of the preprocessor is: a modified version of the main Prolog module (NP); for each CIAO Prolog module m_i, a modified CIAO Prolog version nm_i; and for each ASP module e_i, a CIAO module (im_i) and a class definition (c_i).[2]

The transformation of the Prolog modules consists of a simple rewriting process, used to adapt the syntax of the interface constraints and make it compatible with CIAO Prolog's syntax. For example, the rules passed as arguments to `assert` and `retracts` have to be quoted to allow the peculiarities of ASP syntax to be accepted. The transformation of each ASP module leads to the creation of two entities that will be employed during the actual program execution: an *interface module* and a *model class*. These are described in the following subsections. The preprocessor will also automatically invoke the CIAO Prolog toplevel and load all the appropriate modules for execution. The interaction with the user is the same as the standard CIAO Prolog toplevel.

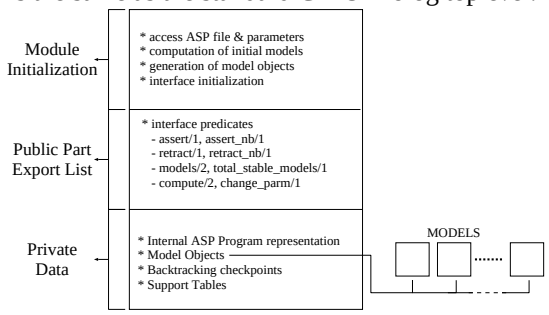

Figure 1: Structure of the Interface Module

Interface Modules The preprocessor generates one interface module for each ASP module present in the original input program. The interface module is implemented as a standard CIAO Prolog module and it provides the client Prolog modules with the predicates used to access and manage the ASP module. The interface module is created for each ASP module by instantiating a generic module skeleton.

The overall structure of the interface module is illustrated in Figure 1. The module has an export list which includes

[2] CIAO provides the ability to define classes and create class instances (Pineda & Bueno 2002).

all the predicates used to manipulate ASP modules (e.g., `assert`, `retract`, `model`) as well as all the predicates that are defined within the ASP module.

The definition of the various exported predicates (except for the predicates defined in the ASP module) is derived by instantiating a generic definition of each predicate. Each module has an initialization part, which is in charge of setting up the internal data structures and invoke the ASP solver (*Smodels*) for the first time on the ASP module. The result of the computation of the models will be encoded as a collection of *Model Objects* (see the description of the Model Classes in the next subsection). The module will maintain a number of internal data structures, including a representation of the ASP code, a representation of the parameters to be used for the computation of the answer sets (e.g., values of constants), a list containing the objects representing the models of the ASP module, a counter of the number of answer sets currently present, etc.

Model Classes The preprocessor generates a CIAO class definition for each ASP module. The objects obtained from the instantiation of such class will be used to represent the individual models of the ASP module. Prolog modules can obtain reference to these objects (e.g., using the `model` predicate supplied by the interface module) and use them to directly query the content of one model. The definition of the class is obtained through a straightforward parsing of the ASP module, to collect the names of the predicates defined in it; the class will provide a public method for each of the predicates present in the ASP module. The class defines also a public method `add/1` which is used by the interface module to initialize the content of the model.

Each model is stored in one instance of the class; the actual atoms representing the model are stored internally in the objects as facts of the form $s(\langle$`fact`$\rangle)$.

Examples

Planning: Let us continue with the example of the blocks-world planning problem. We have three blocks `a, b` and `c`. Initially, block `a` is on block `b`, block `b` is on the table and block `c` is on the table. The goal state is: block `b` is on `c`, block `c` is on `a` and finally block `a` is on the table. The objective is to determine what block moves (represented by facts of the type `move(source,destination,time)`) are required to achieve the goal state—assuming that we can move only one block at a time, and we can move only blocks that are not covered by other blocks. The Prolog module allows the user to

- use the Prolog program to explore the space of possible plans—e.g., if we do not want to accept plans that move block `a` to block `b`, then we can add the goal
  `setof(T,(plan:model(Y,Q),Q:move(a,b,T)),[])`
  which will determine a model (if any) that does not contain any fact of the form `move(a,b,T)`.
- we can perform selection of models according to some quantitative criteria. For example, if we assume that each `moveop` action has a *cost*—i.e., the facts generated during planning have the form `move(source,dest,time,cost)`, then the following code can be used to select the most expensive plan under $5000:

```
setof(X,plan:model(X),List),
   find_plan(List,P,Cost), Cost < 5000.
find_plan([],_,5000).
find_plan([M|Rest],MM,MC) :-
   find_plan(Rest,M1,C1),
   findall(C,M:move(_,_,_,C),Costs),
   sum_list(Costs, Cost),
   ( Cost > C1, Cost < 5000 ->
     MM = M, MC=Cost; MM = M1, MC=C1 ).
```

QA with Multiple Knowledge Sources: Let us elaborate the travel domain QA problem described in (Baral et al. 2005). The problem is to model a travel domain, specify an instance where `john` is traveling from `paris` to `baghdad`, and perform reasoning about consequences of the travel (e.g., location of his laptop depending on whether `john` was arrested during the trip).

Traveling Domain and Commonsense Knowledge: this is encoded in an ASP module (let's call it `trip_know`). It will encode knowledge about flights, how they can be used to get to a destination (through multiple hops), etc.; omitting most details due to lack of space, we can have description of actions:

```
act(embark(F,N)):- flight(F),person(N).
...
```

Similarly, we can describe the fluents, e.g.,

```
fluent(onboard(N,F)):- person(N),flight(F).
...
```

Actions have preconditions and effects, e.g.,

```
h(at(Name,City),T+1):-h(partic(Name,Trip),T),
         occ(disembark(Flight,Name),T),
         exec(disembark(Flight,Name),T),
         destination(Flight,City), time(T).
exec(disembark(Flight,Name),T):-
         action(disembark(Flight,Name)),
         h(en_route(Flight,Dest),T), time(T),
         h(onboard(Name,Flight),T).
```

Static causal laws relate fluents, e.g.,

```
h(at(Obj,City),T) :- h(at(Pers,City),T),
        h(has(Pers,Obj),T), person(Pers),
        obj(Obj),city(City), time(T),
        not ~h(at(Obj,City),T).
```

Various axioms will also be needed, e.g., to address the frame problem:

```
h(F,T+1) :- h(F,T), fluent(F), not ~h(F,T+1).
~h(F,T+1) :- ~h(F,T), fluent(F), not h(F,T+1).
```

At each point in time we expect only one action to occur; using the choice rules of Smodels:

```
1{occ(A)}1 :- action(A).
```

Collection of facts can be used to describe specific problem, instances; e.g., each flight:

```
flight(xyz). depart(xyz,paris).
destination(xyz,baghdad). ...
```

Web Access Module: note that the description of the specific instance of a problem requires two sources of information; one is the description of objects and participants, one is the description of flights. The second source of information, in the real world, is acquired from web sites; let us assume that the description of the flights is stored on the Web, as an XML file, at URL `http://www.xx.xxx/travel.xml`. `travel.xml`, then we can develop a Prolog module that extracts the facts from the XML file and feeds them to the ASP module, e.g., (using CIAO's PiLLoW library for web access and manipulation)

```
get_flight(NAME) :-
 url_info('www.xx.xxx/travel.xml',UI),
 fetch_url(UI,[],R), member(content(C),R),
 xml2terms(R,XML),
 extract_xml_fligh(NAME,XML,Info),
 Info=env(flight,[name=Name],Details),
 trip_know:assert(flight(NAME)),
 member(depart(City1),Details),
 trip_know:assert(depart(NAME,City1)),
 member(arrive(City2),Details),
 trip_know:assert(destination(NAME,City1)),...
```

CLP Module: Similarly to what discussed in (Baral et al. 2005), we can make use of CLP to handle the numeric parts, of the computation, e.g., dates. The answer sets obtained from the ASP module `trip_know` provide "virtual" time steps when different actions occur. The CLP module will need to associate more precise information to this steps—i.e., locate the "real" time when the different actions occur. The computation will be driven by the answer sets produced by the ASP module. E.g., if we have facts describing the duration of each leg (`duration(Flight,Length)`) and layover in each city (`layover(City,Length)`), then we can use CLP to estimate the arrival times at each destination, and answer questions of the type "If an arrest warrant for john is issued on March 15 at 1pm in London, can he make it to Baghdad?", using CLP rules as:

```
safe :-   model(Q),
   compute_arrival(london, Time1,Q),
   Q: layover(london,Time2),
   convert(march,15,13,00,Time3),
   Time1+Time2 #< Time3.
compute_arrival(City,ArriveTime,Q) :-
  Q: start_time(T),
  recur_arrival(paris,City,T,ArriveTime,Q).
recur_arrival(Dest,Dest,T,T,_).
recur_arrival(City,Dest,T1,T2,Q) :-
  Q:layover(City,Lay),
  Q:h(at(john,City),T),
  Q:occ(embark(john,Flight),T),
  Q:destination(Flight,City1),
  Q:duration(Flight,Dur),
  T3 is T1+Lay+Dur,
  recur_arrival(City1,City,T3,T2,Q).
```

Here we assume that time is expressed using an absolute number (and `convert` dates and times into absolute numbers). In turn, the CLP module can explore alternative routes computed by the ASP for desirable properties; for example, if we are seeking whether the traveler can find a route that will allow him to be in `london` on March 15th at 1pm, we can simply write:

```
?- findall(Q,trip_know:model(Q),List),
   member(Model, List),
   compute_arrival(london, Time1,Model),
   Model: layover(london, Time2),
   convert(march,15,13,00,Time3),
   Time1 #<= Time3, Time3 #<= Time1+Time2.
```

Prolog's backtracking will explore the different models (different routes to get to destination) and see if one allows `john` to be in `london` at the desired time.

Conclusion and Future Work

In this paper we presented ASP−PROLOG, a system which provides a tight integration between CIAO Prolog and ASP. The system allows to create programs which are composed of Prolog modules and ASP modules. ASP modules contain either complete or fragments of ASP programs, Prolog modules are capable of accessing ASP modules, to read and/or modify their content—through the traditional Prolog `assert` and `retract` predicates. Prolog modules are also capable of accessing the answer sets of each ASP module, and use them during the execution—e.g., to solve goal against them. The prototype implementation of ASP−PROLOG, built using CIAO Prolog and *Smodels*, is available at `www.cs.nmsu.edu/~okhatib/asp_prolog.html`.

We will continue the development of ASP−PROLOG; in particular, we wish to investigate the possibility of a reverse communication process, where the ASP modules are capable of proactively requesting information from the Prolog modules—an investigation is in progress to allow ASP modules to make use of CLP capabilities.

References

Baral, C. et al. 2005. Textual Inference by Combining Multiple Logic Programming Paradigms. In *AAAI Workshop on Inference for Textual Question Answering*.

Eiter, T. et al. 1998. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In KR'98, 406–417.

Gelfond, M., and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programs. In ICLP, 1070–1080. MIT Press.

Harabagiu, S. 2001. Just-In-Time Question Answering. In *NLPRS*, 27–34.

Hermenegildo, M., et al. 1999. The CIAO Multi-Dialect Compiler and System. In *Parallelism and Implementation of Constraint Logic Programming*. Nova Science. 65–85.

Lloyd, J. 1987. *Foundations of Logic Progr*. Springer.

Niemela, I. 1999. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and AI*, 25:241–273.

Pasca, M. 2000. Open-domain Factual Answer Extraction. In *Technical Report, SMU*.

Pineda, M., and Bueno, F. 2002. The O'Ciao Approach to Object Oriented Logic Programming. In *Colloquium on Implementation of Constraint Logic Programming Systems*.

Vicedo, J.L. 2000. A Semantic Approach to Question Answering Systems. *TREC-9*, 440-445.