Towards a CLP System for Reasoning about Answer Sets

O. Elkhatib E. Pontelli T.C. Son
New Mexico State University
Department of Computer Science
{okhatib,epontell,tson}@cs.nmsu.edu

Abstract

We present a system ($\mathbb{ASP} - \mathbb{PROLOG}$) which provides a tight and well-defined integration of Prolog and Answer Set Programming (ASP). The combined system enhances the expressive power of ASP, allowing us to write programs that *reason* about *dynamic* ASP modules and about collections of stable models. These features are vital in a number of application domains (e.g., planning, scheduling, diagnosis). We describe the design of $\mathbb{ASP} - \mathbb{PROLOG}$ along with its implementation, realized using CIAO Prolog and *Smodels*.

1 Introduction

Stable model semantics [4] is a widely accepted approach to provide semantics to logic programs with negation. Stable model semantics relies on the idea of accepting multiple minimal models as a description of the meaning of a program. In spite of its wide acceptance and its extensive mathematical foundations, stable models semantics have only recently found its way into "practical" logic programming. The recent successes have been sparked by the availability of efficient inference engines (such as *Smodels* [13], Cmodels [7], ASSAT [9], and DLV [3]) and a substantial effort towards *understanding* how to write programs under stable models semantics [12, 11, 8]. This has led to the development of a novel *programming paradigm*, commonly referred to as *Answer Set Programming (ASP)*. ASP is a computation paradigm in which logical theories (Horn clauses with negation) serve as problem specifications and solutions are represented by *collection of models*. ASP has been concretized in a number of related formalisms—e.g., disjunctive logic programming [3]. In comparison to other non-monotonic logics, ASP is syntactically simple and, at the same time, very expressive. ASP has been adopted in various domains (e.g., [8, 5, 14]).

Most existing ASP inference engines have been extended to provide front-ends that are suitable to encode different types of knowledge. *Smodels* provides a rich set of built-in structures to express choices, weight-constraints, and restricted forms of optimizations. *DLV* provides different classes of constraint rules (e.g., weak constraints), aggregates, and alternative front-ends (e.g., diagnosis, planning), allowing the development of programs in specific applications domains using very high-level languages. 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 and batch process. Most existing ASP systems offer only a batch approach to execution of programs—programs are completely developed, they go through a "compilation" process, executed and finally stable models are proposed to the user. The process lacks of any level of interaction with the user. In particular, it does not directly 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 programmers can control the computation of stable models through the rules that they include in the logic program. Nevertheless, ASP systems offer very limited capabilities for reasoning on the *whole class* of stable models associated to a program—e.g., to perform selection of models according to user-defined criteria or to compare across models. These activities are very important in many application domains—e.g., to express soft constraints on models, to support preferences when using ASP to perform planning.
- ASP systems are independent systems; interaction with other languages can be performed only through low level and complex APIs; this prevents programmers from writing programs that manipulate ASP programs and stable models as first-class citizens. We would like to be able to write programs in a high-level language (Prolog in this case), which are capable to access ASP programs, modify their structure (by adding or removing rules and facts), and access and reason with stable models. This type of features is essential in many ASP applications. For example, ASP planners require to pre-specify the maximum length of the plan; the ability to access and modify ASP programs would allow us to write programs that automatically modify the length of the plan until a plan with the desired property is found.

In this project we propose a system, called $\mathbb{ASP}-\mathbb{PROLOG}$. The system 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. $\mathbb{ASP}-\mathbb{PROLOG}$ 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 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 models, and modify it (using the traditional `assert` and `retract`).

We are not aware of any system with the same capabilities as $\mathbb{ASP}-\mathbb{PROLOG}$. Relatively limited work has been presented exploring effective ways of integrating ASP in the context of other programming languages. *Smodels* provides a very low level API [17] which allows C++ programs to use *Smodels* as a library. DLV does not document any external API, although a Java wrapper has been recently announced [1]. XASP [2] proposes an interface from XSB to the API of *Smodels*. It provides a subset of the functionalities of $\mathbb{ASP}-\mathbb{PROLOG}$, with a deeper integration with the capabilities of XSB of handling normal logic programs.

2 Brief Semantic Foundations

In this section, we discuss the semantic foundation of $\mathbb{ASP}-\mathbb{PROLOG}$ and motivate the basic constructions of the language. For simplicity, we will assume a pure Prolog system, though in the real systems, full-blown Prolog will be allowed.

2.1 Language Formalization

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 that $\mathcal{F}_A \subseteq \mathcal{F}_P$ and $\mathcal{F}_A$ is finite.

- Π is a set of predicates; in particular, $\Pi = \Pi_P \cup \Pi_A \cup \Pi_C$, where $\texttt{true}, \texttt{false} \in \Pi_P \cap \Pi_A$ and
  - Π_P are called *user-defined* predicates;
  - Π_A are called *ASP-defined* predicates;
  - Π_C are called *Interface* predicates. In this presentation we will limit our attention to $\Pi_C = \{\texttt{assert}, \texttt{retract}, \texttt{models}\}$.
- $\mathcal{F}_A \cup \Pi_A \subseteq \mathcal{F}_C$.

The function ar determines the arity of the various symbols. 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 (i.e., *Prolog*) and A (i.e., *ASP*). The language should meet the following requirements:

- 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 $\mathsf{A}^{ar(f)} \rightarrow \mathsf{A}$ ($\mathsf{A}^{ar(f)}$);
- the symbols in $\mathcal{F}_A$ and Π_A are of sort A and P at the same time.

Intuitively, 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 $\mathcal{F}_P \cup \mathcal{V}$ and Π_C is called an *Interface-atom*.

Definition 1 *An* ASP-literal *is either an ASP-atom or a formula of the type not A, where A is an ASP-atom. An* ASP clause *is a rule of the form*

$$A \;{:-}\; L_1 \wedge \ldots \wedge L_n \tag{1}$$

$${:-}\; L_1 \wedge \ldots \wedge L_n \tag{2}$$

where A is a ground ASP-atom, and $L_1, \ldots, L_n$ are ground ASP-literals. Rules of type (2) are known as constraint rules*.*

Definition 2 (ASP constraint) *An* ASP constraint *is a formula of the type $L_1 \wedge \ldots \wedge L_k$, where $k \geq 0$ and each L_i is*

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

Definition 3 (Interface Constraints) *An* Interface constraint *is a conjunction $L_1 \wedge \ldots \wedge L_k$ ($k \geq 0$) of interface atoms of the type*

$$\texttt{assert}(A \;{:-}\; B_1, \ldots, B_n) \quad \texttt{retract}(A \;{:-}\; B_1, \ldots, B_n) \quad \texttt{models}(t)$$

where $A \;{:-}\; B_1, \ldots, B_n$ is an ASP clause and t is a P-term.

Definition 4 ($\mathbb{ASP}-\mathbb{PROLOG}$ rule) *A $\mathbb{ASP}-\mathbb{PROLOG}$ rule is a formula of the form*

$$H \;{:-}\; C_1, C_2 \; [] \; B_1, \ldots, B_k$$

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 (or, simply, a static rule) is a $\mathbb{ASP}-\mathbb{PROLOG}$ rule that does not contain any interface constraint based on `assert` *or* `retract`*.*

Definition 5 ($\mathbb{ASP}-\mathbb{PROLOG}$ program) *A $\mathbb{ASP}-\mathbb{PROLOG}$ 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 $\mathbb{ASP}-\mathbb{PROLOG}$ program is a $\mathbb{ASP}-\mathbb{PROLOG}$ program $\langle Pr, As\rangle$ such that all the rules in Pr are static.*

E.g., the following is an ASP clause: $p(a) :- q(a) \wedge r(b)$ where p, q, r are in Π_A and a, b are in $\mathcal{F}_A$.

2.2 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` operations in the interface constraints 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 [12]; let us denote with $\mathcal{M}(As) = \{M \subseteq \mathcal{B}_A \mid M \text{ is a stable model of } As\}$.

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 $\langle M, \nu\rangle$ is a model of the program if $M \subseteq \mathcal{B}_P$ and it satisfies all the Pr rules; in particular, the model will satisfy a ground 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 an 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 straightforward to extend these definitions to deal with entailment of an arbitrary goal and to define when clauses are satisfied by the model. Observe that, given a program $P = \langle Pr, As\rangle$ and a fixed model naming function ν, we have that there exists 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 a top-down operational semantics.

Definition 6 (Update) *Given a program $P = \langle Pr, As\rangle$ and an interface constraints p, we define the* update of P w.r.t. p *(i.e., $\mathcal{U}(P,p)$) as follows:*

$$\mathcal{U}(P,p) = \begin{cases} \langle Pr, As \cup \{r\}\rangle & \text{if } p = \texttt{assert}(r) \\ \langle Pr, As \setminus \{r\}\rangle & \text{if } p = \texttt{retract}(r) \end{cases}$$

Let R be a *computation rule* [10]—i.e., a function $R : \mathcal{B}^* \rightarrow \mathcal{B}$ which is used to select a subgoal; in particular we will denote with R_{Prolog} the computation rule that selects the leftmost subgoal.

Definition 7 (State) *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 $\mathcal{H}$*

[1] For the sake of simplicity we focus on a single ASP module; the presentation can be easily generalized to accommodate multiple ASP modules.

- *τ is a 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 (i.e., the function such that forall $X \in \mathcal{V}.\epsilon(X) = X$), and τ_0 is the function that is undefined for every input.*

The notion of entailment is defined through a transition relation between states.

Definition 8 (Derivation Step) *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:

- $R(G) = A$
- *if A is a P-atom, then there exist 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 exists 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 exists 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 exists 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*
  - $|K| = |\mathcal{M}(As')|$
  - *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 $\mathcal{M}(As)$*
  - $\tau' = \tau \circ \{s_1 \mapsto S_1, \ldots, s_r \mapsto S_r\}$

Definition 9 (Entailment) *Given a program $P = \langle Pr, As \rangle$ and a goal G, we say that $P \models G\sigma$ iff $\langle G, \epsilon, \tau_0, As \rangle \vdash_R^* \langle \emptyset, \sigma, \tau, As' \rangle$.*

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

The $\mathbb{ASP} - \mathbb{PROLOG}$ system has been developed as an extension of the CIAO Prolog system [6]. 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). The handling of the ASP modules is left to the *Smodels* system [13].

3.1 Concrete Syntax

The abstract syntax presented in the previous section has been refined in the $\mathbb{ASP} - \mathbb{PROLOG}$ system 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 contains an ASP program. We will use an ASP program—called `plan.pl`—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.pl` consists of rules specifying the initial configuration (e.g., all blocks on the table), the goal configuration (e.g., all blocks stacked in a given order), 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 an input parameter called *steps* that determines the (maximal) length of the plan. A call to this program looks like `lparse -c steps=5 plan.pl | smodels 0` which will return all stable models of `plan.pl`, each corresponds to a plan of length 5. We will now detail the syntax of $\mathbb{ASP} - \mathbb{PROLOG}$.

3.1.1 Module Interface

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

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

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 parameters with their values, to be passed from the Prolog module to the ASP module.

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

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

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

3.1.2 Interface Constraints

We have provided predicates that allow Prolog modules to query and manage ASP modules:

- `model/2`: in $\mathbb{ASP} - \mathbb{PROLOG}$ models of an ASP module can be retrieved using indices; the `model` predicate relates an index number to the term representing the corresponding model. The `model` predicate has to be qualified with the ASP module on which it is meant to be applied. E.g., the goal `plan:model(1, Q)` will allow a Prolog module to access the first model of the module `plan.pl`. More precisely, variable Q will be instantiated with the first model of `plan.pl`. The goal will fail if the program `plan.pl` does not have a stable model.[2]
- `total_stable_model/1`: the predicate is satisfied if the argument is the number of models of the ASP module. For example, `plan:total_stable_model(X), X>0` will succeed if `plan.pl` has at least one stable model and fails otherwise.

[2] `model` is a simplified version of `models/1` described earlier.

- `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 a ASP-constraint that prevents the occurrence of the action $move(a, table)$. From a Prolog module, we can issue

  ```
  assert(plan:[(:-move(a, table, T), time(T))])
  ```

  which will add the constraint "`:-move(a, table, T), time(T).`" to `plan.pl`.
- `assert_nb/1` and `retract_nb/1`: the $\mathbb{ASP}-\mathbb{PROLOG}$ system provides also an alternative version of the `assert` and `retract` predicates. The main difference is that the modifications derived from `assert` and `retract`, as illustrated in the semantics description in Section 2, will be undone during backtracking, while the modifications to an ASP module performed using `assert_nb` and `retract_nb` will remain unaffected by 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 parameters dynamically. The following Prolog fragment allows us to change the *steps* parameter of `plan.pl`:

  ```
  blocks_solve :- plan:total_stable_models(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, the predicate `chk_condition` will check whether a plan satisfies certain condition or not (see below) 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`, then we can issue the goal `plan : compute(3,_).`
- `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* `plan` *contains the following rules defining the predicate* `p`*:*

  $$p(a) :- q(a), r(a). \qquad\qquad p(b) :- r(b).$$

  Then the Prolog goal `plan:clause(p(X), Y)` *has two solutions:*

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

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

3.1.3 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 a stable model 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), display('move '), display(X),
                   display(' on '), display(Y), display(' at time '),
                   display(T), nl, fail.
print_sol(_, _).
```

3.2 System Implementation

The overall structure of the implementation is depicted in Figure 1. The system is composed of two parts, a *preprocessor* and the actual CIAO Prolog system.

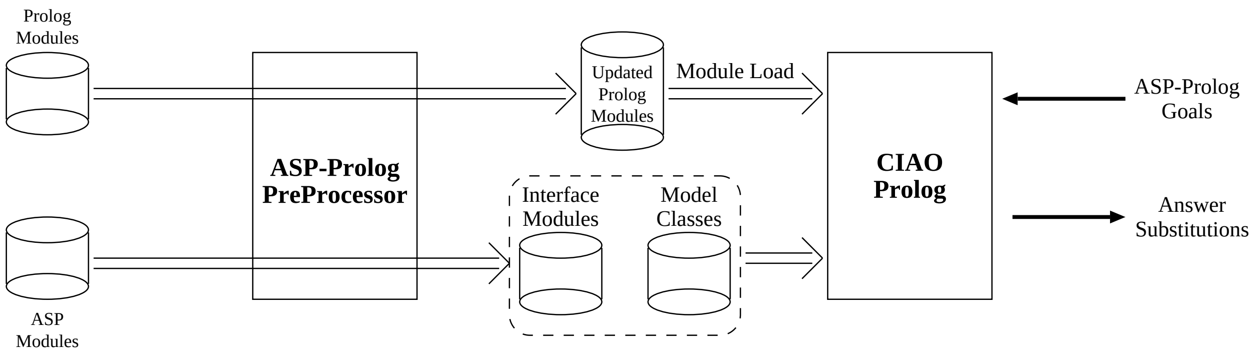

Figure 1: Overall Structure of $\mathbb{ASP}-\mathbb{PROLOG}$ Implementation

3.2.1 Preprocessing

The input to the preprocessor is composed of *(i)* the main Prolog module (Pr); *(ii)* a collection of CIAO Prolog modules ($m_1, m_2, \ldots, m_n$); *(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), a modified version of the other Prolog modules ($nm_1, nm_2, \ldots, nm_n$), and for each ASP module e_i the preprocessor creates a CIAO module (im_i) and a class definition (c_i).[3]

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

[3] CIAO provides the ability to define classes and create class instances [15].

example, the rules passed as arguments to `assert` and `retracts` have to be quoted to allow the peculiarities of ASP syntax (e.g., the use of braces for choice rules) 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 that of the standard CIAO Prolog toplevel.

3.2.2 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 to the content of the specific ASP module considered.

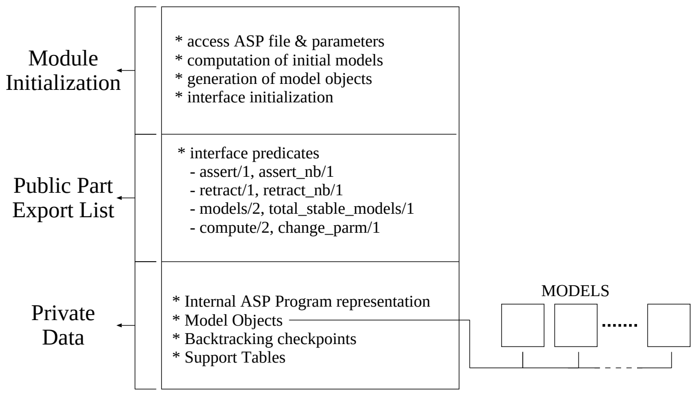

Figure 2: Structure of the Interface Module

The overall structure of the interface module is illustrated in Figure 2. The module has an export list which includes 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.[4] The typical module declaration generated for an interface module will look like:

```
:- module('t23.xxx', [ assert/1, retract/1,
                       assert_nb/1, retract_nb/1,
                       model/2, change_parm/1, compute/2,
                       total_stable_model/1,
                       p/0, q/0, r/0 ]).
```

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 (e.g., the internal representation of the ASP module, tables to store parameters and stable models), and invoke the answer set solvers for the first time on the ASP module—in the current prototype we are using *Smodels* as answer set solver. The result of the computation of the models will be encoded as a

[4]Due to a limitation in the current implementation of CIAO's module system, we cannot dynamically add new predicates to an existing ASP module—as CIAO does not support, yet, dynamic redefinition of a module.

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 stable models (e.g., values of constants), a list containing the objects representing the models of the ASP module, a counter of the number of stable models currently present, etc.

3.2.3 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. In addition, the class defines also a public method `add/1` which is employed 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 $s(\langle\texttt{fact}\rangle)$.

For instance, if we have a simple ASP module containing the rules:

$$\texttt{p :- q.} \qquad \texttt{q :-} \ r. \qquad \texttt{r.}$$

then the preprocessor will generate a class definition of the type:

```
  :- class(t23_class).
  :- dynamic s/1.      %% used to store the facts of the model
%% export declarations for the ASP predicates
  :- export(p/0).
  :- export(q/0).
  :- export(r/0).
%% utility method for building the model
  :- export(add/1).
%% definition of the methods
  p :- s(p).
  q :- s(q).
  r :- s(r).
%% add a new element to the model
  add(X) :- assertz_fact(s(X)).
```

3.3 Implementation Details

Interface Predicates: The various interface predicates are implemented in CIAO Prolog in a fairly straightforward way. Some general observations:

- The implementation of `assert` proceeds by adding the new rules to the module and recomputing the models; the structure of the main clause implementing it is

  ```
  assert(L) :- assert1(L),
          module_concat('t23.xxx', assert2(L), M), und(M).
  assert2(L) :- \+ empty_list(L), retract_nbf(L).
  ```

  The `module_concat` and `und` are internal predicates of CIAO Prolog that allows us to specify what action to take upon backtracking through the clause; in this case, `assert2` will be called upon backtracking, which will undo the modifications and restore the previous set of models. `assert_nb` will avoid the final step—since changes will not be undone during backtracking.

- The implementation of `retract` follows a similar structure; rules are removed (if they are present) from the module and the models are recomputed accordingly. The modifications are cached to ensure undoing upon backtracking. The main clauses implementing it are:

```
retract(L) :- \+ empty_list(L), !, retract1(L),
        store_list_rr(L1),
        module_concat('t23.xxx', retract2(L1), M), und(M).
retract2(L) :- \+ empty_list(L), assert_nb(L).
```

  The `retract1` performs the modification of the module and the recomputation of the models; `store_list_rr` places the modifications in the trail structure; the final declarations in the `retract` rule indicate what predicate should be call upon backtracking—`retract2`. As we can see, `retract2` simply restores the rules that have been previously removed (using `assert_nb`), and restores the original set of models.
- the same structure can be found in the implementation of `compute`; if called with arguments unbound, then the predicate will access the current `compute` configuration (i.e., it will indicate how many models have been requested and whether there is a core of literals that have to be true in every model); if called with bound arguments, having a value different then the current `compute` configuration, then the models will be recomputed with the new configuration. As for `assert` and `retract`, the `compute` will set up a hook to allow for undoing effect of the changes during backtracking.

Internal Data Structures: A number of tables are maintained by each interface module to support the execution of ASP modules. Some of the relevant internal structures include:

- *fn:* used to maintain a (Prolog-based) representation of the rules composing the ASP module;
- *uf:* a temporary table aimed at supporting the process of rules unification during execution of `assert` and `retract`;
- *stable_ref:* a table (implemented as Prolog facts) that maintain references to the current models of the ASP module (as pairs *model number*/*object reference* that maps name of models to objects representing the models);
- *retract_rule:* a trail structure that caches the modifications performed by `assert` and `retract`; this is required to allow undoing of the chances;
- *prm:* a table (encoded as Prolog facts) that stores the parameters to be used during the computation of the models of the ASP module.

4 Examples

Let us continue with the example of the planning problem. The planner is aimed at computing the movements of blocks from initial state to a goal state. 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($source, destination, time, cost$) then we can select the plan with the lowest cost by writing

```
      ...
      setof([X,Y], plan:model(X,Y), List),  %% collect all models
      find_smallest_plan(List,P,Cost).
find_smallest_plan([[Index,Model]], Model, Cost) :-
      findall(C, Model:move(_,_,_,C), Costs),
      sum_list(Costs,Cost).
find_smallest_plan([ [Index,Model] | Rest], MinModel, MinCost) :-
      find_smallest_plan(Rest,M1,C1),
      findall(C,Model:move(_,_,_,C),Costs),
      sum_list(Costs, Cost),
      ( Cost < C1 -> MinModel = Model, MinCost=Cost;
                     MinModel = M1, MinCost=C1 ).
```

5 Conclusion and Future Work

In this paper we presented $\mathbb{ASP}-\mathbb{PROLOG}$, a system which provides a tight and semantically well-founded integration between Prolog (in the form of CIAO Prolog) and answer set programming (in the form of *Smodels*). 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, expressed using the *lparse* input language [18]. 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 stable models of each ASP module, and use them during the execution—e.g., to solve goal against them. At the syntax level, $\mathbb{ASP}-\mathbb{PROLOG}$ guarantees the same style of programming and syntax as traditional Prolog programming, integrating ASP modules and stable models as first-class citizens of the languages. $\mathbb{ASP}-\mathbb{PROLOG}$ allows to extend the expressive power of ASP, allowing to write Prolog programs that can dynamically modify ASP modules, reason about stable model, and promotes incremental and 'what-if' approaches to the construction of ASP programs.

The prototype implementation of $\mathbb{ASP}-\mathbb{PROLOG}$, built using CIAO Prolog and *Smodels*, is available at `www.cs.nmsu.edu/~okhatib/asp_prolog.html`. We will continue the development of $\mathbb{ASP}-\mathbb{PROLOG}$ by:

- using $\mathbb{ASP}-\mathbb{PROLOG}$ in the development of various ASP applications where an interactive environment is more appropriate; and
- investigating the possibility of a reverse communication process, where the ASP modules are capable of proactively requesting information from the Prolog modules—an investigation in this direction is in progress to allow ASP modules to make use of CLP capabilities [16].

Acknowledgements

This work is partially supported by NSF grants CCR9875279, CCR9900320, CDA9729848, EIA0130887, EIA9810732, and HRD9906130.

References

[1] The DLV Wrapper Project. `160.97.47.246:8080/wrapper`, 2003.

[2] L. Castro, T. Swift, and D.S. Warren. *XASP: Answer Set Programming with XSB and Smodels.* SUNY Stony Brook, 2002. `xsb.sourceforge.net/packages/xasp.pdf`.

[3] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *KR-98*, pages 406–417, 1998.

[4] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *ICLPS-88*, pages 1070–1080. MIT Press, 1988.

[5] K. Heljanko and I. Niemela. Answer Set Programming and Bounded Model Checking. In *TPLP*, 3(4):519–550, 2003.

[6] M. Hermenegildo et al. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[7] Y. Lierler and M. Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In *LPNMR'04*,pages 346–350, 2004.

[8] V. Lifschitz. Action Languages, Answer Sets, and Planning. In *The Logic Programming Paradigm.* Springer Verlag, 1999.

[9] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. In *AAAI*, pages 112–117, 2002.

[10] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Heidelberg, 1987.

[11] V.W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm.* Springer Verlag, 1999.

[12] I. Niemela. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and AI*, 25(3/4):241–273, 1999.

[13] I. Niemela and P. Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In *LPNMR-97*, pages 421–430, 1997.

[14] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog Descision Support System for the Space Shuttle. In *PADL-01*, pages 169–183, 2001.

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

[16] T.C. Son and E. Pontelli. Planning with preferences using logic programming. In *LPNMR'04*, pages 247–260, 2004.

[17] T. Syrjänen. Lparse User's Manual. `http://www.tcs.hut.fi/Software/smodels/`.

[18] T. Syrjänen. Implementation of Local Grounding for Logic Programs with Stable Model Semantics. Technical Report B-18, Helsinki University of Technology, 1998.