Integrating an Answer Set Solver into Prolog: ASP – PROLOG

Omar Elkhatib, Enrico Pontelli, and Tran Cao Son

Department of Computer Science,
New Mexico State University
{okhatib, epontell, tson}@cs.nmsu.edu

1 Introduction

A number of answer set solvers have been proposed in recent years, such as *Smodels*, DLV, Cmodels, and ASSAT. Most existing ASP solvers have been extended to provide front-ends that are suitable to encode specialized forms of knowledge—e.g., weight-constraints, restricted forms of optimization, front-ends for planning and diagnosis. These features allow declarative solutions in specific application domains. However, this is not completely satisfactory:

- The development of an ASP program is viewed as a "monolithic" process. Most ASP systems offer only a batch approach to execution of programs—programs are completely developed, "compiled", executed, and finally answer sets are proposed to the user. The process lacks any levels of interaction with the user. In particular, it does not directly support an interactive development of programs (as in Prolog), where one can immediately explore the results of simply adding/removing rules.
- ASP programmers can control the computation of answer sets through the rules that they include in the logic program. Nevertheless, 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 or to compare models. These activities are important in many application domains—e.g., to express soft constraints, to support preferences when using ASP to perform planning.
- ASP solvers are independent systems; interaction with other languages can be performed only through complex, low level APIs; this prevents programmers from writing programs that manipulate ASP programs and answer sets as first-class citizens. E.g., we wish 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 access and reason with answer sets. This type of features is essential in many domains—e.g., automatically modify the plan length in a planning problem.

We address these problems by developing a system, ASP – PROLOG. The system provides a tight integration of ASP in Prolog. The language is developed using the module and class capabilities of CIAO Prolog. ASP – PROLOG allows

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 399–404, 2005.

programmers to assemble a variety of different modules to create a program; along with the traditional types of modules supported by CIAO Prolog (e.g., standard Prolog, constraint logic programming, active deductive databases), it allows the presence of various *ASP modules*, each being a logic program conforming to the syntax of **lparse**. Each Prolog module can access any ASP modules (using the traditional *module qualification* of Prolog), read its content, access its models, and modify it (e.g., adding/removing rules).

2 System Capabilities and Possible Areas of Application

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

:– `use_asp(`*`module_name, file_name, parameters`*`)`

where *module_name* is the name of the ASP module, *file_name* is the file containing the ASP code, while *parameters* control the ASP solver (command line and `compute` arguments of SMODELS). $\mathbb{ASP} - \mathbb{PROLOG}$ provides predicates that allow Prolog to interact with ASP modules:

- `model(ModelName,ModelObject)` retrieves one answer set of an ASP module; `ModelName` is an atom uniquely identifying one answer set, while `ModelObject` is a CIAO Prolog object containing the answer set (as Prolog facts). For example, `plan:model(1, Q)` retrieves the answer set named `1` of ASP module `plan` and stores it as an object in `Q`; if we want to check whether the atom `p` is true in such answer set, we simply execute the Prolog goal `Q:p`.
- *`total_stable_model/2`* determines the number of answer sets of an ASP module, and returns a list of the names given to the answer sets.
- `assert/1` and `retract/1`: the argument of these predicates is a list of ASP rules, that are either added or removed from an ASP module. For example, the goal `plan:assert([p:-q])` adds the rule `p:-q` to the ASP module `plan`. Modifications are undone during backtracking.
- `assert_nb/1` and `retract_nb/1` have the same effect as `assert`/`retract`, with the exception that the modifications are not undone upon backtracking.
- `change_parm/1` allows to set/modify the parameters for the ASP execution (e.g., values of constants, components of the `compute` statement of SMODELS).
- `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.

If α is a CIAO object representing an answer set, then the Prolog goal $\alpha : p$ corresponds to testing truth of p (possibly non-ground) in the answer set α. Observe that, due to the fact that the syntax of *Smodels* is not ISO Prolog-compliant, certain *Smodels* constructs (e.g., cardinality and weight constraints) have a slightly different syntactic representation when used within Prolog modules.

Possible Application Areas: ASP – PROLOG allows users to

- *manipulate answer sets of a program*; this includes *(i)* the computation of the entailment relation of a logic program (e.g., different modes of reasoning, such as *skeptical* or *credulous* reasoning can be done), and *(ii)* comparing answer sets of an ASP program, and select those that satisfy certain properties (e.g., individual preferences).
  Since ASP has frequently been used as a knowledge representation language, the ability to compute the entailment relation of a logic program makes ASP – PROLOG an attractive candidate for the implementation of query-answering systems based on answer set semantics. Furthermore, because computing preferred answer sets has found its application in planning with preferences, diagnosis, and common-sense reasoning, ASP – PROLOG can be used as an interactive environment for such systems.
- *modify programs and recompute their answer sets on the fly* (e.g., adding or removing rules); this feature provides a simple way to modify the values of constants occurring in a program and/or to delay the grounding process of a program with variables until the instantiated rules are needed. Furthermore, ASP – PROLOG provides different ways for users to test and debug an ASP program interactively. For example, 'suspected rules' can be removed for testing the consistency of a program; adding 'known-to-be-true' atoms into a program is another way for detecting errors in the program; etc.
- *work with several logic programs simultaneously* (e.g., reasoning about the common knowledge of multiple-agents); since ASP – PROLOG allows users to work with several modules at the same time, it can be used as an environment for implementing/testing various formalisms for modeling multi-agents. For example, when each agent's knowledge is represented by a logic program, computing common knowledge among them is equivalent to computing the intersection of all possible answer sets.
- use logic programs with answer set semantics to control the query-answering process of a Prolog program (e.g., avoiding infinite loops); this is possible, since ASP – PROLOG programs are Prolog programs extended with a new type of atoms, whose truth value is determined by the ASP solver.

We are not aware of any systems with the same capabilities as ASP – PROLOG. *Smodels* provides a very low level API [5] that 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 ASP – PROLOG, with a deeper integration with the capabilities of XSB of handling normal logic programs.

3 System Implementation

The syntax and semantics of ASP – PROLOG programs are described in [3]. It suffices to notice that a ASP – PROLOG program is a pair (Pr, As), where Pr is a set of ASP – PROLOG rules and As is a logic program which conforms to

the syntax of **lparse**. Each $\mathbb{ASP}-\mathbb{PROLOG}$ rule is a Prolog rule whose body can contain atoms occurring in As. Furthermore, the above mentioned interface predicates (e.g., `assert, retract`) can be used by $\mathbb{ASP}-\mathbb{PROLOG}$ programs to manipulate As. 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. The system accepts the input 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 ($e_1, e_2, \ldots, e_m$).

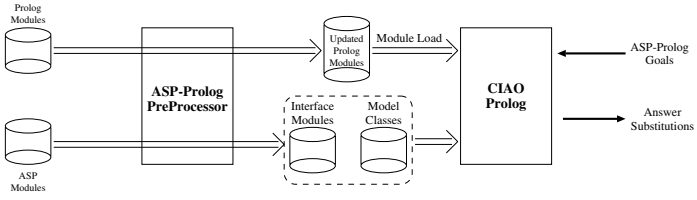

Fig. 1. Overall Structure of $\mathbb{ASP}-\mathbb{PROLOG}$ Implementation

Preprocessing: The input to the system is used as the input to the preprocessor. The preprocessor transforms each Prolog module into a new module (Pr is transformed to NPr and m_i is transformed to nm_i), and each ASP module e_i into a CIAO module im_i and a class definition c_i.[1] The main purpose of this step is to adapt the syntax of the interface predicates, make it compatible with CIAO Prolog's syntax, and prepare different *objects* (interface module and model class) for the actual program execution. The preprocessor also invokes the CIAO Prolog top-level and loads all the appropriate modules for execution. The interaction with the user is the same as the standard Prolog top-level.

Interface Modules: For each ASP module e_i, the preprocessor generates an interface module c_i by instantiating a generic module skeleton to the content of e_i. c_i is a standard CIAO Prolog module and provides the client Prolog modules with the predicates used to access and manage the ASP module e_i. 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. The typical module declaration generated for an interface module will look like:

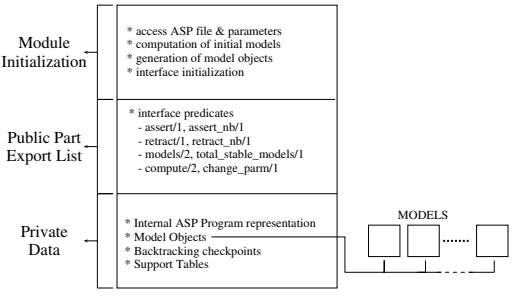

Fig. 2. Structure of the Interface Module

[1] CIAO provides the ability to define classes and create class instances [4].

`:- module('plan.lp',[assert/1,retract/1,...,model/2,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 answer sets), 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 collection of *Model Objects* (see below). 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 of the objects representing the models of the ASP module, and a count of the current number of answer sets.

Model Classes: For each ASP module e_i, the preprocessor generates a CIAO class definition c_i. 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 or several models. 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 $s(\langle \texttt{fact} \rangle)$.

Implementation and System Details: The various interface predicates are implemented in CIAO Prolog in a fairly straightforward way. For instance, the implementation of `assert` (resp. `retract`) makes use of the `module_concat` of CIAO Prolog to introduce new (resp. remove) rules to (resp. from) the ASP module.

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:* maintains a (Prolog-based) representation of the rules of the ASP module;
- *stable_ref:* a table (implemented as Prolog facts) that maintains references to the current answer sets of the ASP module (as pairs *model name*/*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 changes;
- *prm:* a table (encoded as Prolog facts) that stores the parameters to be used during the computation of the models of the ASP module.

Code and URL: The original idea and the semantics of $\mathbb{ASP} - \mathbb{PROLOG}$ has been presented in [3]. The first version of system is now complete and available for download at `www.cs.nmsu.edu/~okhatib/asp_prolog.html`.

References

1. The DLV Wrapper Project. `160.97.47.246:8080/wrapper`, 2003.
2. L. Castro. XASP: Answer Set Programming with XSB. SUNY Stony Brook, 2002.
3. O. Elkhatib et al. ASP-Prolog: A System for Reasoning about Answer Set Programs in Prolog. In *PADL-2004*, pages 148–162. Springer, 2004.
4. M. Pineda et al. The O'Ciao Approach to Object Oriented Logic Programming. In *Colloquium on Implementation of Constraint Logic Programming Systems*, 2002.
5. T. Syrjänen. Lparse User's Manual. `www.tcs.hut.fi/Software/smodels`.