A Language for Modular Answer Set Programming: Application to ACC Tournament Scheduling

Luis Tari, Chitta Baral, and Saadat Anwar

Arizona State University
Department of Computer Science and Engineering
Brickyard Suite 501, 699 South Mill Avenue, Tempe, AZ 85287
{luis.tari,chitta,saadat.anwar}@asu.edu

Abstract. In this paper we develop a declarative language for modular answer set programming (ASP). Our language allows to declaratively state how one ASP module can import processed answer sets from another ASP module. We define the syntax and semantics of our language and illustrate its applicability by modeling the ACC tournament scheduling problem. Besides the elegance of developing declarative programs in a modular manner, our illustration shows that a problem that is not timely solvable when done in a monolithic way, but becomes solvable when done in a modular way.

1 Introduction

One of the common answer set programming methodologies is to use logic programming with answer set semantics (also referred to as AnsProlog or A-Prolog) to describe a problem so that the answer sets of the program encode solutions to the problem. In this approach the programs usually have two parts: one which enumerates possible solutions and another part which tests and eliminates non-solutions. Both of these parts are written using AnsProlog rules, and they are given together to an answer set solver such as Smodels [14], DLV [5] or ASSAT [12]. Although none of these solvers explicitly enumerate each possible answer set and test them one by one, the size of the state space of the problem does play a role in terms of the time taken to find solutions. When one is unable to solve a problem using the above approach in a reasonable amount of time one wonders if somehow the representation can be factored and solved in parts.

For example consider an AnsProlog program whose rules can be divided into four parts: the enumeration parts E_1, E_2, and the test parts T_1 and T_2. Let us assume that the program $T_1 \cup E_1 \cup T_2 \cup E_2$ can be split into the bottom $T_1 \cup E_1$ and top $T_2 \cup E_2$ [11]. In this case one can find the answer set of the program by first computing the answer sets of $T_1 \cup E_1$, and for each answer set A_i of $T_1 \cup E_1$, compute the answer sets of $T_2 \cup E_2 \cup A_i$. Thus if E_1 enumerates n binary atoms and E_2 enumerates m binary atoms, and $T_1 \cup E_1$ has only a small number k of solutions, then directly trying to find answer sets of $T_1 \cup E_1 \cup T_2 \cup E_2$ could involve potentially dealing with a search space of 2^{m+n}, while the above mentioned factored approach would lead to searching in one potential search space to 2^n, and k potential search spaces of 2^m.

Although the existing answer set solvers are able to take advantage of the above to some extent, the extent to which they do is not clear. An alternative approach would be that the programmer specifies the above mentioned factoring and orders the problem solving system to solve it in a factored manner. Unfortunately, until now no such system based on answer set programming exists. One can write a Prolog program (or a program in another language) which will make multiple calls to the answer set solver and solve the problem in a factored manner. We believe several people have taken this route. *In this paper we propose an alternative. We propose a declarative language that builds on AnsProlog and allows declarative specification of such factoring as well as two other processing steps that one often encounters when solving large problems by breaking down the problem and making multiple calls to answer set solvers.*

We refer to our methodology as modular answer set programming (modular ASP). Besides the efficiency with respect to many problems, as in other programming languages, modules promote reusability of code. Large programs can be broken down into modules to improve readability of code. We consider the ACC basketball tournament scheduling problem [13] which has a large state space. We encoded two approaches of the ACC scheduling problem and solved using Smodels, DLV, ASSAT and CMODELS [10]. Our code is available at http://www.public.asu.edu/∼tng01/modules_asp.html. However, none of them was able to return a solution in 30 minutes. On the other hand, we were able to get solutions using our proposed modular ASP system by breaking the problem into modules.

The rest of the paper is organized as follows. In Section 2 we present the syntax and semantics of modular ASP programs, and the implementation. In Section 3 we illustrate the uses of the various import rules with some practical examples. In Section 4 we illustrate how we solve the ACC basketball tournament scheduling problem [13] using modular ASP. In Section 5 we discuss related work and in Section 6 we conclude.

2 Modular ASP programs

In this section we define the syntax and semantics of modular ASP programs. We borrow the traditional terminology of atoms, and literals.

2.1 Syntax

Definition 1 An ASP rule is of the form:

$a_0 \leftarrow a_1, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n.$

where a_is are atoms. □

Definition 2 An ASP import rule is one of the following three forms:

(a) $q(\bar{X})$:- $M(\bar{b}).p(\bar{X}, \bar{a})$.

(b) $*q(\bar{X})$:- $M(\bar{b}).p(\bar{X}, \bar{a})$.

(c) $q(\#, \bar{X})$:- $M(\bar{b}).p(\bar{X}, \bar{a})$.

where q and p are predicate names, M is a module name, $\bar{X}$ is a tuple of variables, $\bar{a}$ and $\bar{b}$ are tuple of constants, and $*$ and $\#$ are special symbols. □

Intuitively, rule (a) says that q imports the set of tuples $\bar{X}$ from the module $M(\bar{b})$ such that $p(\bar{X}, \bar{a})$ are true in all answer sets of $M(\bar{b})$. Unlike rule (a), rule (b) imports a set of set of tuples $\mathcal{S}$ about the predicate q. Each imported set of tuples S corresponds to an answer set S' of the module $M(\bar{b})$. Each atom $q(\bar{X})$ in S in $\mathcal{S}$ owes its presence to the presence of $p(\bar{X}, \bar{a})$ in S'. Rule (c) is similar to rule (b), but unlike the latter, it numbers the answer sets of $M(\bar{b})$ in a predefined way. The tuple $(i, \bar{X})$ is then imported to q if $p(\bar{X}, \bar{a})$ is true in the i-th answer set of $M(\bar{b})$.
In the above ASP import rules, $p(\bar{X}, \bar{a})$ is evaluated with respect to the module M.

Definition 3 An ASP module consists of:
(i) a name, (ii) a set of parameters, (iii) a collection of ASP rules, and (iv) a collection of ASP import rules. □

Definition 4 If P is the name of an ASP module and the parameters of P are $\{X_1, \ldots, X_n\}$, then $P(X_1 = a_1, \ldots, X_n = a_n)$, or simply $P(a_1, \ldots, a_n)$, denotes the ASP module obtained by replacing X_i with a_i in all the AnsProlog rules of P, and is referred to as an *instantiation* of P. □

Example 1 Suppose P is the name of an ASP module and P has the following rule:

```
date(N).
```

$P(N = 9)$ is an instantiation of P which indicates that the value of N becomes 9 in module P. □

Definition 5 A modular ASP program is a collection of ASP modules where each module has a different name. □

Earlier we define the import rules individually. If there are multiple import rules in a module, we need to ensure certain consistencies. This is precisely defined in the next subsection.

2.2 Semantics

Given a modular ASP program $\{M_1, \ldots, M_l\}$ its dependency graph is constructed as follows. The nodes of the graph are the modules and there is an edge from M_i to M_j if M_j has an import rule with M_i in its body. In this paper we are only concerned with the semantics of modular ASP programs whose dependency graph does not have cycles. In addition, for import rules that import from the same module, we require that they have to be of the same form. We define the answer sets of ASP modules with respect to a modular ASP program that contains it. For simplicity, we may just mention answer sets of ASP modules, and the corresponding modular ASP program is understood from the context.

Definition 6 Given an ASP import rule r of the form (b) and Y as an answer set of $M(\bar{b})$, we say X is obtained by filtering Y using r if $X = \{q(\bar{c}) : p(\bar{c}, \bar{a}) \in Y\}$. □

Definition 7 Given a set of ASP import rules R = { r_1, . . ., r_n } that are of the same form (a) or (b) or (c), and each r_i contains module name $M(\bar{b})$, the extracted set of R is a set of sets of literals obtained as follows:

(i) If all import rules in R are of the form (a), then for each r_i, compute S_i = { $q(\bar{c})$: $p(\bar{c}, \bar{a})$ is in all answer sets of $M(\bar{b})$}. The extracted set of R is a singleton set consisting of the set S = $S_1 \cup \ldots \cup S_n$.

(ii) If all import rules in R are of the form (b), let $Y_1, \ldots, Y_m$ be the answer sets of $M(\bar{b})$. Then for each r_i, compute S_{ij} = $\{X : Y_j$ is an answer set of $M(\bar{b}),$ and X is obtained by filtering Y_j using $r_i\}$. The extracted set of R is the set S = { S_1, . . ., S_m }, where S_j = $S_{1j} \cup \ldots \cup S_{nj}$.

(iii) If all import rules in R are of the form (c), the extracted set of R is obtained as follows:

– Compute the extracted set S assuming the rules are of the form (b).
– The number of elements in the set S is at least 1. If the set S is a singleton set consisting of the empty set then the extracted set of R is S. Otherwise,
  - Order each element of S by an ordering criteria and label them as $S_1, \ldots, S_m$.
  - The extracted set of R is the set { S' }, where $S' = \{q(j, \bar{c}) \mid q(\bar{c}) \in S_j\}$.

□

Example 2 Suppose $P1$ is the following ASP module:

```
%%=== module P1 ===%%
p(c,d,a).
p(e,f,a) :- not r(g,h,a).
r(g,h,a) :- not p(e,f,a).
```

The module $P1$ has two answer sets:

$\{p(c, d, a), p(e, f, a)\}$ and $\{p(c, d, a), r(g, h, a)\}$

Suppose $P2$ is an ASP module that contains import rules R = { r_1, r_2 } of the form (a) that refer to $P1$.

```
%%=== module P2  ===%%
%%import rule r1
q(X,Y) :- P1.p(X,Y,a).
%%import rule r2
s(X,Y) :- P1.r(X,Y,a).
```

The predicate $p(c, d, a)$ exists in all answer sets of $P1$. Therefore, the extracted set of R is $\{\{q(c, d)\}\}$. □

Example 3 Suppose $P3$ is an ASP module that contains import rules R = { r_1, r_2 } of the form (b) that refer to $P1$, which is defined in Example 2.

```
%%=== module P3 ===%%
%%import rule r1
*q(X,Y) :- P1.p(X,Y,a).
%%import rule r2
*s(X,Y) :- P1.r(X,Y,a).
```

By filtering the answer sets of $P1$ using R, the extracted set of R is $\{\{q(c,d), q(e,f)\}$, $\{q(c,d), s(g,h)\}\}$. □

Example 4 Suppose $P4$ is an ASP module that contains import rules R = { r_1, r_2 } of the form (c) that refers to $P1$, which is defined in Example 2.

```
%%=== module P4 ===%%
%%import rule r1
q(#,X,Y) :- P1.p(X,Y,a).
%%import rule r1
s(#,X,Y) :- P1.r(X,Y,a).
```

The extracted set of R is
$\{\{q(1,c,d), q(1,e,f), q(2,c,d), s(2,g,h)\}\}$. □

We now define the answer set of instantiated ASP modules in two steps: first for the ones without import rules, and next for the ones with import rules.

Definition 8 Let $P(X_1 = a_1, \ldots, X_n = a_n)$ be an instantiated ASP module with no import rules. The answer sets of $P(X_1 = a_1, \ldots, X_n = a_n)$ are the answer sets of the logic program consisting of the rules of $P(X_1 = a_1, \ldots, X_n = a_n)$. □

Definition 9 Let $P(X_1 = a_1, \ldots, X_n = a_n)$ be an instantiated ASP module with the set of import rules $R = \{r_1, \ldots, r_l\}$ such that no two elements of R that import from the same module are of different forms. Let $\{M_1(\bar{b_1})$, ..., $M_k(\bar{b_k})$ } be a set of unique occurrences of module names appearing in all import rules of $P(X_1 = a_1, \ldots, X_n = a_n)$ and $R_{M_i(\bar{b_i})} \subseteq R$ be the import rules containing the module name $M_i(\bar{b_i})$. X is an answer set of $P(X_1 = a_1, \ldots, X_n = a_n)$ computed by replacing $R_{M_i(\bar{b_i})}$ from $P(X_1 = a_1, \ldots, X_n = a_n)$ by the atoms in an extracted set of $R_{M_i(\bar{b_i})}$. □

Example 5 In example 4, the extracted set of atoms for import rules $r1$ and $r2$ are added to module $P4$. Therefore, module $P4$ has the answer set { $q(1,c,d)$, $q(1,e,f)$, $q(2,c,d)$, $s(2,g,h)$ }. □

2.3 Implementation

We implemented a prototype of the above defined modular ASP system. The prototype is composed of two parts: a front-end that interprets the modular AnsProlog language and the Smodels [14] inference engine as the backbone.

The input to the system is a set of ASP modules and the output is the answer sets computed by Smodels. The front-end interpreter scans for any import rules and proceeds as follows. Assume that the input is a set of ASP modules m_1 and m_2, and m_1 has an import rule that refers to m_2. The interpreter first scans for any import rule in m_1. In this case, the interpreter finds an import rule that refers to module m_2 and looks into m_2 for any other import rules. The backbone of the system then computes the answer sets of a module once no further import rule is found. The answer sets of m_2 are then stored as a temporary file and the import rules in m_1 are then replaced by the necessary predicates in m_2 to form a new temporary AnsProlog program $m_1{}'$. The answer sets of $m_1{}'$ are displayed as the output of the system.

The frontend interpreter was written in Perl and tested under the Cygwin environment. Smodels version 2.27 was used as the backbone of our system. We tested our prototype with the implementation of the ACC basketball tournament scheduling problem.

3 Practical Examples to Illustrate Import Rules

In this section, we illustrate the uses of the various import rules based on a mini basketball tournament scheduling problem.

Suppose SCH_GEN is an ASP module that generates valid schedules for a mini basketball tournament that involves four teams: clem, duke, fsu and geog. Each team plays each other twice during the course of the tournament. The module SCH_GEN has a parameter that defines the number of days for the tournament.

Example 6 $SCH\_GEN(num\_days = 3)$ is an instantiation of SCH_GEN, which indicates that the value of num_days becomes 3. □

Assume that the above instantiation of SCH_GEN generates two schedules:

$$\{sch(duke, clem, 1), sch(fsu, geog, 1), sch(duke, fsu, 2), sch(clem, geog, 2), \\ sch(geog, duke, 3), sch(fsu, clem, 3)\}$$

and

$$\{sch(duke, clem, 1), sch(geog, fsu, 1), sch(geog, duke, 2), sch(fsu, clem, 2), \\ sch(duke, fsu, 3), sch(clem, geog, 3)\}$$

The predicate $sch(X, Y, Z)$ means that team X plays with team Y at home on date Z.

Example 7 Suppose $P1$ is the following ASP module that finds which team duke must play with on a particular day. This can be done by importing the valid schedules related to duke from module SCH_GEN through import rule r of form (a). The predicate $duke\_must\_play(Y, D)$ indicates that duke must play with team Y on day D. The idea is that if all valid schedules agree that team Y plays with duke on day D, then it must be the case that team Y plays with duke on day D.

```
%%=== module P1 ===%%
%% import rule r
duke_must_play(Y,D) :- SCH_GEN(num_days=3).sch(duke,Y,D).
```

The predicate $sch(duke, clem, 1)$ exists in all answer sets of SCH_GEN. Therefore, the extracted set of r is $\{duke\_must\_play(clem, 1)\}$. □

Example 8 Suppose $P2$ is the following ASP module that finds all valid schedules for day 1. The idea is to import the valid schedules related to day 1 from module SCH_GEN through import rule r of form (b). The predicate $play\_on\_day1(X, Y)$ indicates that team X plays with team Y at home on day 1.

```
%%=== module P2 ===%%
%% import rule r
*play_on_day1(X,Y) :- SCH_GEN(num_days=3).sch(X,Y,1).
```

The extracted set of r is { $\{play\_on\_day1(duke, clem), play\_on\_day1(fsu, geog)\}$, $\{play\_on\_day1(duke, clem), play\_on\_day1(geog, fsu)\}$ }. □

Example 9 Suppose $P3$ is the following ASP module that finds a schedule in which duke plays at home on 2 consecutive days. This can be done by importing the valid schedules related to duke from module SCH_GEN through import rule r of form (c). The predicate $duke\_plays(N, Y, D)$ indicates that duke plays with team Y on day D according to the N-th schedule, while the predicate $good\_sch(N)$ indicates that the N-th schedule includes duke playing at home on 2 consecutive days.

```
%%=== module P3 ===%%
%% import rule r
duke_plays(#,Y,D) :- SCH_GEN(num_days=3).sch(duke,Y,D).
good_sch(N) :- team(Y), team(YY), day(D),
  duke_plays(N,Y,D), duke_plays(N,YY,DD), Y!=YY, DD=D+1.
```

The extracted set of r is { { $duke\_plays(1, clem, 1)$, $duke\_plays(1, fsu, 2)$ }, { $duke\_plays(2, clem, 1)$, $duke\_plays(2, fsu, 3)$ } }. So $good\_sch(1)$ is true. □

4 An Illustration of the Use of Modules

We use the scheduling problem of the Atlantic Coast Conference (ACC) college basketball tournament to motivate the use of modules. We adopted the problem description given in [7]. The ACC basketball tournament scheduling problem is a round-robin scheduling problem. Nine teams from nine universities compete with each other over a period of nine weeks.

In the ACC schedule, there were two games per week. One on a weekday and the other on a weekend, giving a total of eighteen game days. On each day, eight of the teams play either a home game or an away game, while the ninth team has a bye, i.e. the ninth team does not play. The schedule is a double round-robin in which every team plays against every other team twice - once at home and once away. To reflect the fact that the schedule is a double round-robin, a mirroring scheme is used to group dates into pairs (r_1, r_2) such that each team plays against the same team in dates r_1 and r_2. We adopted the mirroring scheme m [13], by assigning m={(1,8), (2,9), (3,12), (4,12), (5,14), (6,15), (7,16), (10,17), (11,18)}. The following rules are used to describe the domain of the ACC tournament scheduling problem:

```
% Game type): home (h), away (a), bye (b)
gtype(h;a;b).

% A period of nine weeks and two games per week:
% one on a week day and the other on a weekend.
date(1..18).        wend(2;4;6;8;10;12;14;16;18).

% The nine teams in the ACC tournament
team(clem;duke;fsu;gtech;umd;unc;ncst;uva;wake).

% Rival teams
rival(clem,gtech).  rival(duke,unc).
rival(umd,uva).     rival(ncst,wake).

% Mirroring scheme:
% first group (mgp_dates1) has mirroring after 9 dates;
% second (mgp_dates2) has mirroring after 7 dates.
mgp_dates1(3..7). mgp_dates2(1;2;10;11).
```

Various constraints are imposed on the schedule in order to maintain fairness and maximize profits. We divide the constraints into three main categories, as in [7]:

1. **Generic team scheduling constraints**, e.g. no team may have more than two away matches in a row.
2. **Individual team scheduling constraints**, e.g. Duke has a bye on date 16.
3. **Rival team scheduling constraints**, e.g. UNC plays its rival Duke on the last date and on date 11.

In the following subsections, we describe three different approaches in solving the scheduling problem: monolithic, mirroring and modular approaches. The monolithic approach is based on the intuitive idea of generating all possible schedules among the teams and then prune schedules that are not compatible to the constraints of the problem. To scale down the search space, we utilized the mirroring scheme of dates in the mirroring approach [7]. We implemented both monolithic and mirroring approaches on popular answer set solvers such as Smodels [14], DLV [5], ASSAT [12] and CMODELS [10]. The solvers do not return the answer sets within 30 minutes on a Pentium IV 2.8 GHz computer with 512 MB of RAM, as the search space becomes too large for both approaches.

To reduce the complexity, we adopted the breakdown strategy given in [7], which we refer to as the modular approach, and implemented it using the modular ASP language.

4.1 Monolithic Approach

We first demonstrate the monolithic approach by showing the Smodels code for the generation of possible schedules. Due to the limit of space, the code for the team-specific constraints is not shown. The predicate $sch(X, Y, D)$ implies that team X plays against team Y at home on date D, while the predicate $play(X,Y,D)$ means that teams X and Y play against each other on date D.

```
%% can only play at most one other team at home
0 { sch(X,Y,D) : team(Y) } 1 :- date(D), team(X).

%% cannot play more than 1 team on a particular date
:- team(X;Y;Z), date(D), Y!=Z, play(X,Y,D), play(X,Z,D).

%% cannot play both home and away on same date
:- team(X;Y), date(D), sch(X,Y,D), sch(Y,X,D).

%% cannot play itself
:- team(X;Y), date(D), sch(X,Y,D), X==Y.

%% The schedule should be such that a team plays at home
%% with another team exactly once
1 { sch(X,Y,D) : date(D) } 1 :- team(X), team(Y), X!=Y.

%% Each team must have 8 home games, 8 away games and 2
%% byes over the course of the season.
cond_home :-  8 {plays_at_home(X,D) : date(D)} 8, team(X).
:- not cond_home.
cond_away :-  8 {plays_away(X,D) : date(D)} 8, team(X).
:- not cond_away.
cond_bye :-  2 {has_bye(X,D) : date(D)} 2, team(X).
:- not cond_bye.
```

The encodings for DLV, ASSAT and CMODELS were done without the use of cardinality constraints. To replace the cardinality constraints, predicates for counting described in [1] were used. Counting was used for finding the number of home, away games and byes for a team. Specifically, $count\_home(X,N)$ implies that for a team x, $count\_home(x,n)$ is true if there are n different facts of $sch(x,y,d)$ with distinct ys. The predicate $count\_away(X,N)$ implies for a team x, $count\_away(x,n)$ is true if there are n different facts of $sch(y,x,d)$ with distinct ys. The predicate $count\_bye(x,n)$ is true if there are n different facts of $has\_bye(x,d)$ for team x with distinct ds. Due to space limitation, only code for generation of possible schedules is shown below.

```
%% Generate all possible schedules
sch(X,Y,D) :- team(X), team(Y), date(D), not n_sch(X,Y,D).
n_sch(X,Y,D) :- team(X), team(Y), date(D), not sch(X,Y,D).

%% Each team must have 8 home games, 8 away games and
%% 2 byes over the course of the season.
:-  team(X), number(N), count_home(X,N), N>9.
:-  team(X), number(N), count_home(X,N), N<8.
:-  team(X), number(N), count_away(X,N), N>9.
:-  team(X), number(N), count_away(X,N), N<8.
:-  team(X), number(N), count_bye(X,N),  N>3.
:-  team(X), number(N), count_bye(X,N),  N<2.
```

The generation of all possible schedules among the teams becomes infeasible, due to the number of variables and the range of the values of the variables. In particular, the number of atoms for the predicate $sch(X, Y, Z)$ is $9 \times 9 \times 18$, as there are 9 teams and 18 game dates. So the number of interpretations for the $sch(X, Y, Z)$ atoms is $2^{9 \cdot 9 \cdot 18}$ of $sch(X, Y, Z)$. Smodels, DLV, ASSAT and CMODELS were used but no solution was returned within 30 minutes. To reduce the space complexity, we attempted the mirroring approach described in the next subsection.

4.2 Mirroring Approach

The mirroring approach imposes an ordering on half of the schedule dates. We first demonstrate the mirroring approach by showing the Smodels code for the generation of possible schedules using the mirroring scheme of dates. Due to the limit of space, the code for the scheduling constraints is not shown.

```
% The schedule should be such that for a particular date,
% a team may only play at most one team at home.
0 { sch(X,Y,D) : team(Y) } 1 :- mgp_dates1(D), team(X).
0 { sch(X,Y,D) : team(Y) } 1 :- mgp_dates2(D), team(X).

% Teams assigned to play during the dates in mgp_dates1(D)
% and mgp_dates2(D) will play again after 9 and 7 dates.
sch(Y,X,D+9) :- sch(X,Y,D), team(X), team(Y), mgp_dates1(D).
sch(Y,X,D+7) :- sch(X,Y,D), team(X), team(Y), mgp_dates2(D).

plays(X,Y,D) :- team(X), team(Y), date(D), sch(X,Y,D).
plays(X,Y,D) :- team(X), team(Y), date(D), sch(Y,X,D).
plays(X,D) :- team(X), team(Y), date(D), plays(X,Y,D).
plays_at_home(X,D) :- team(X), team(Y), date(D), sch(X,Y,D).
plays_away(X,D) :- team(X), team(Y),date(D), sch(Y,X,D).
has_bye(X,D) :- team(X), date(D), not plays(X,D).
```

Similar to the monolithic approach, counting predicates were used in place of cardinality constraints for the encodings of DLV, ASSAT and CMODELS. The code for the generation of schedules is shown below:

```
%% Generation of schedules using mirroring dates
sch(X,Y,D) :- team(X;Y), mgp_days1(D), not nsch(X,Y,D).
nsch(X,Y,D) :- team(X;Y), mgp_days1(D), not sch(X,Y,D).
sch(X,Y,D) :- team(X;Y), mgp_days2(D), not nsch(X,Y,D).
nsch(X,Y,D) :- team(X;Y), mgp_days2(D), not sch(X,Y,D).
```

```
%% Use mirroring constraints to generate game patterns.
sch(X,Y,DD) :- sch(Y,X,D), mgp_days1(D), DD=D+9, team(X;Y).
sch(X,Y,DD) :- sch(Y,X,D), mgp_days2(D), DD=D+7, team(X;Y).

%% Each team has 8 home, 8 away and 2 byes per season.
:-  team(X), number(N), count_home(X,N), N>9.
:-  team(X), number(N), count_home(X,N), N<8.
:-  team(X), number(N), count_away(X,N), N>9.
:-  team(X), number(N), count_away(X,N), N<8.
:-  team(X), number(N), count_bye(X,N),  N>3.
:-  team(X), number(N), count_bye(X,N),  N<2.
```

The generation of all possible schedules among the teams becomes infeasible, due to the number of variables and the range of the values of the variables. In particular, the number of atoms for the predicate $sch(X, Y, Z)$ is $9 \times 9 \times 9$, as there are 9 teams and 9 mirroring game dates. So the number of interpretations for the $sch(X, Y, Z)$ atoms is $2^{9 \cdot 9 \cdot 9}$ of $sch(X, Y, Z)$. This can be the reason that the encoding of the mirroring approach using Smodels, DLV, ASSAT and CMODELS did not return the solutions within 30 minutes.

4.3 Modular Approach

In the modular approach, the scheduling problem is broken down into three phases, namely pattern generation, pattern-team binding and timetable generation. We define a pattern to be a combination of home, away, and byes assigned to a particular date. A pattern-team binding is an assignment of a suitable pattern to each of the nine teams. A timetable is built from a pattern-team binding by determining which team plays against which opponent on a given date. A valid timetable must satisfy all constraints. Each of the three phases is implemented as modules. The three corresponding modules are pattern generation `pat_gen`, pattern-team binding `pat_team_bind` and timetable generation `team_team_sch`. We describe each of the main parts of the modules in the remaining of this subsection.

The pattern generation phase is to generate patterns in the form of the predicate $sch(GT, D)$, which refers to a pattern of game type GT to be scheduled on date D. The possible patterns are then pruned by the constraints of the problem, such as “no more than two home games can be played in a row”. The following code demonstrates some of the constraints involved in the generation of patterns for the pattern generation module `pat_gen`:

```
%%% === pat_gen: module pattern generation === %%%
% Schedule a type of game (home,away,bye) on a given date
% to generate patterns. Types of games assigned on dates
% in mgp_dates1(D) and mgp_dates2(D) will play again after
% 9 and 7 dates resp.
1{sch(G,D) : gtype(G)}1 :- mgp_dates1(D).
sch(a,D+9) :- sch(h,D), mgp_dates1(D).
sch(h,D+9) :- sch(a,D), mgp_dates1(D).
sch(b,D+9) :- sch(b,D), mgp_dates1(D).

1{sch(G,D) : gtype(G)}1 :- mgp_dates2(D).
sch(a,D+7) :- sch(h,D), mgp_dates2(D).
sch(h,D+7) :- sch(a,D), mgp_dates2(D).
sch(b,D+7) :- sch(b,D), mgp_dates2(D).

% No team has more than two home games in a row.
:- date(D;D+1;D+2), sch(h,D), sch(h,D+1), sch(h,D+2).

% Exclude patterns that have more than
% 8 home games or 8 away games or 2 byes
% on a given date.
:- 9 { sch(h,D) : date(D) }.
:- 9 { sch(a,D) : date(D) }.
:- 3 { sch(b,D) : date(D) }.
```

Once the valid sets of patterns are generated, the goal of the next phase pattern-team binding is to assign teams to patterns, subject to the team-specific constraints. An example of a team-specific constraint is “Wake does not play home on date 17”. The sets of patterns are denoted by the predicate $pat(N, GT, D)$ from the assignment of numbers N to each of the set of patterns $sch(GT, D)$, imported from the module `pat_gen`. The binding of a team and a pattern is represented by the predicate $bind(T, P)$, where P is a pattern that satisfies the team-specific constraints of team T. The following code illustrates the main part of pattern-team binding module `pat_team_bind` that involves some of the team-specific constraints and the import instructions from module `pat_gen`:

```
%%% === pat_team_bind: module pattern-team binding === %%%
% Import patterns generated by module pat_gen
pat(#,D,G) :- pat_gen.sch(G,D).
npat(P) :- pat(P,G,D).

% Pattern P is bound to team T if bind(T,P).
bound(T,P) :- team(T), bind(T,P), npat(P).

% Home-away pairs
ha_pair(h,a). ha_pair(a,h).

% Pattern P1 has already been bound to the team T
% so each team can only be bound once.
o_bound(T,P) :- team(T), npat(P;P1), P!=P1, bound(T,P1).

% Likewise, team T2 has already been bound to the
% pattern P, so each pattern can only be bound once.
```

```
a_bound(T,P) :- team(T;T2), npat(P), T!=T2, bound(T2,P).

% bind "duke" to a pattern P, such that
% - duke has a bye on date 16
% - unc plays its rival duke on dates 11 and 18
%   i.e. duke does not have a bye on these dates
% - duke does not have away games on date 18
% - neither duke nor P has been bound
bind(duke,P) :- pat(P,16,b), pat(P,18,GT1), GT1!=b,
    pat(P,11,GT2), GT2!=b, pat(P,18,GT3), GT3!=a,
    not o_bound(2,P), not a_bound(2,P).

% number of byes on each game date has to be 1
:- 2 {bound(T,P):team(T):pat(P,D,b)}, date(D).

% number of home games must be 4
:- 1 {bound(T,P):team(T):pat(P,D,h)} 3, date(D).
:- 5 {bound(T,P):team(T):pat(P,D,h)}, date(D).
```

The final phase is the timetable generation, in which team-team schedules are generated based on the patterns and bindings generated by the previous two phases. Patterns $pat(N, D, GT)$ and pattern-team binding $bound(T, N)$ generated by the previous two modules `pat_gen` and `pat_team_bind` are imported to the timetable generation module `team_team_sch`. The goal of the timetable generation phase is to bind feasible team pairings. Suppose teams T_1, T_2, T_3 follow the patterns $\{$(H,1),(H,2),(A,3)$\}$, $\{$(A,1),(H,2),(H,3)$\}$, $\{$(A,1),(A,2),(H,3)$\}$ in the first three dates, then T_1 can play at home against either T_2 or T_3 but not both. The possible team pairings are then checked against the team-team constraints, such as "no team plays in two consecutive dates against Duke and UNC," so that only schedules that conform to the constraints can be in the answer sets. The following code illustrates the main parts of the timetable generation module `team_team_sch` and some of the team-specific constraints:

```
%%%=== team_team_sch: module timetable generation ===%%%
% Import the valid sets of patterns from module pat_gen
pat(#,D,G) :- pat_gen.sch(G,D).
npat(P) :- pat(P,D,G).

% Import team-pattern binding from module pat_team_bind
*bound(T,P) :- pat_team_bind.bound(T,P).

% Generate bindings of who plays whom on date D:
% T1 plays home, T2 plays away. Neither T1 or T2 is
% playing another team on date D.
plays(T1,T2,D) :- date(D), team(T1;T2), bound(T1,P1),
   bound(T2,P2), pat(P1,D,h), pat(P2,D,a),
   not o_plays(T1,T2,D).
```

```
o_plays(T1,T2,D) :- date(D), team(T1;T2;T3), T2!=T3,
   bound(T1,P1), bound(T3,P3), pat(P1,D,h),
   pat(P3,D,a), plays(T1,T3,D).
o_plays(T1,T2,D) :- date(D), team(T0;T1;T2), T0!=T1,
   bound(T0,P0), bound(T2,P2), pat(P0,D,h),
   pat(P2,D,a), plays(T0,T2,D).
```

The encoding of the modular approach using our modular ASP language returns a solution within 20 minutes.

5 Related Work

The notion of modules in logic programming first appeared in [2]. However, it is known that the approach has problems in handling negations [4]. The use of generalized quantifiers was proposed in [4] to incorporate external functions into logic programs in the stable model semantics. It is not clear how import rules of form (b) and (c) can be expressed using generalized quantifiers.

In terms of implementation, popular Prolog inference systems such as Sicstus [9] and XSB [16] have the notion of modules. There are currently a few systems available that have a similar goal to extend AnsProlog to be modular. The language DLP^T [8] seems to be the closest to our language, which extends DLV to allow modules in the form of templates. The notion of templates is analogous to functions in procedural programming languages. However, it is not clear if it can handle import rules of form (c). XASP [3] is a package of the Prolog inference engine XSB that provides an interface to Smodels, so that an AnsProlog program can be executed through a Prolog program. XASP provides the mechanism for Prolog to handle multiple answer sets. ASP-Prolog [6] extends the idea of XASP by adding constructs that allow an easier approach to develop AnsProlog programs. Not only that AnsProlog rules can be accessed in a Prolog module, AnsProlog rules can be easily added or removed from a Smodels program through a Prolog program. By integrating Prolog to be on top of Smodels, both XASP and ASP-Prolog achieve the goal of extending AnsProlog to be modular. However, developing an AnsProlog application in a modular approach now becomes the task of developing a Prolog application. DLV provides a Java wrapper [15] that allow disjunctive logic programs to be used in Java. While Java is an object-oriented language, enabling Java to execute AnsProlog programs has the effect of modularizing AnsProlog programs.

6 Conclusion

We described a logic programming language that extends AnsProlog to incorporate modules in the form of AnsProlog and Prolog. The main emphasis of our system is that we extend AnsProlog to be modular without inventing an entirely new programming language. Unlike the other systems, the extension of AnsProlog to incorporate modules is done without integrating AnsProlog to an existing language. With the modular ASP language, developing an AnsProlog application in a modular approach is not very different from developing a monolithic AnsProlog program. Not only that our system is able to import answer sets from another module, it can also import predicates specified in other modules. The introduction of the export declarations and import rules is analogous to the idea of member variable access in object-oriented languages. Predicates declared as export in the modular ASP language are similar to the idea of public variables, while predicates that are not declared as export are similar to private variables. This provides control of the level of access of predicates, which is vital to the development of large applications.

It is known that the current implementation of the inference engines for AnsProlog is not efficient in handling numbers. As a future work, we would like to extend our modular ASP language to incorporate modules written in Prolog. The incorporation of Prolog modules can compensate the inability of effectively dealing with numbers and numerical computations in the current AnsProlog inference engines.

References

1. Chitta Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.
2. Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994.
3. Luis Castro, Terrance Swift, and David S. Warren. Xasp: Answer set programming with xsb and smodels. *xsb.sourceforge.net/packages/xasp.pdf*, 2002.
4. Thomas Eiter, Georg Gottlob, and Helmut Veith. Generalized quantifiers in logic programs. In *ESSLLI '97: Revised Lectures from the 9th European Summer School on Logic, Language, and Information*, pages 72–98, London, UK, 2000. Springer-Verlag.
5. Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The kr system dlv: Progress report, comparisons and benchmarks. In *Proceedings Sixth Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.
6. Omar Elkhatib, Enrico Pontelli, and Tran Cao Son. Asp-prolog: a system for reasoning about answer set programs in prolog. In *NMR*, pages 155–163, 2004.
7. Martin Henz. Scheduling a major college basketball conference—revisited. *Operations Research*, 49(1):163–168, 2001.
8. Giovambattista Ianni, Giuseppe Ielpa, Francesco Calimeri, Adriana Pietramala, and Maria Carmela Santoro. Enhancing answer set programming with templates. *In Proceedings of the 10th International Workshop on Non-Monotonic Reasoning NMR2004*, pages 233–239, 2004.
9. S. Kista. Sicstus prolog. *SICStus Prolog User's Guide*, 1990.
10. Y. Lierler and M. Maratea. Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In *Lect. Notes in Comput. Sci.*, volume 2923, pages 346–350. Springer, Berlin, 2004.
11. Vladimir Lifschitz and H. Turner. Splitting a logic program. *In Proceedings of the Eleventh Int'l Conf. on Logic Programming*, pages 23–38, 1994.
12. Fangzhen Lin and Yuting Zhao. Assat: computing answer sets of a logic program by sat solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.
13. George L. Nemhauser and Michael A. Trick. Scheduling a major college basketball conference. *Oper. Res.*, 46(1):1–8, 1998.
14. Ilkka Niemel$\ddot{a}$ and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. pages 421–430. Springer Verlag, 1997.

15. Francesco Ricca. A java wrapper for dlv. *Answer Set Programming*, 2003.
16. Konstantinos Sagonas, Terrance Swift, David S. Warren, Juliana Freire, and Prasad Rao. Xsb prolog. *The XSB System Version 2.2 Volume 1: Programmer's Manual*.