Macros, Macro calls and Use of Ensembles in Modular Answer Set Programming

Chitta Baral, Juraj Dzifcak and Hiro Takahashi
{chitta,juraj.dzifcak,hiro}@asu.edu

Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287

Abstract. Currently, most knowledge representation using logic programming with answer set semantics (AnsProlog) is 'flat'. In this paper we elaborate on our thoughts about a modular structure for knowledge representation and declarative problem solving formalism using AnsProlog. We present language constructs that allow defining of modules and calling of such modules from programs. This allows one to write large knowledge bases or declarative problem solving programs by reusing existing modules instead of writing everything from scratch. We report on an implementation that allows such constructs. Our ultimate aim is to facilitate the creation and use of a repository of modules that can be used by knowledge engineers without having to re-implement basic knowledge representation concepts from scratch.

1 Introduction

Currently, most knowledge representation languages are 'flat'. In other words, for the most part they are non-modular. (It is often mentioned that CYC's language [Guha 1990] allows the use of modules. But this is not well published outside CYC.) Our focus in this paper is the knowledge representation language AnsProlog [Gelfond & Lifschitz 1988,Baral 2003] (logic programming with answer set semantics), where most programs are a collection of AnsProlog rules. Although sets of AnsProlog rules in these programs are often grouped together with comments that describe the purpose of those rules, the existing syntax does not allow one to construct libraries of modules that can be used in different programs. Such libraries are commonplace in many of the programming languages such as C++ and Java and recently in domains such as natural language [Miller *et al.* 1990]. The presence of such libraries makes it easier to write large programs without always starting from scratch, by referring and using already written pieces of code (modules, methods, subroutines etc.).

There are many other advantages of using libraries of modules. For example, having higher level modules available enforces code standardization. A module repository also has the benefit of being proven over the years and hence deemed reliable. In addition, modules may be built using multiple languages which lends to an overall application architecture where strengths of a language are fully exploited without having to find a work-around.

There are several ways to introduce modularity into answer set programming. Some of the ways to do that include:

(1) Macros: Modules are defined as macros or templates. A macro-call would be replaced by a collection of AnsProlog rules as specified by a semantics of the macro call. Such an approach with focus on aggregates is used in [Calimeri *et al.* 2004].

(2) Procedure/method calls: A module is an AnsProlog program with well defined input and output predicates. Other programs can include calls to such a module with a specification of the input and a specification of the output. Such an approach is used in [Tari, Baral, & Anwar 2005].

(3) Procedure/method calls with a specified engine: Here a module is also an AnsProlog program with not only well-defined input and output predicates, but also with an associated inference engine. For example, the associated engine could be a top-down engine such as Prolog or Constraint logic programming, or an answer set enumerator such as Smodels or DLV. Such an approach with respect to constraint logic programming can be built on the recent work [Baselice, Bonatti, & Gelfond 2005] where answer set programming is combined with constraint logic programming.

In this paper, we will focus on the first way to represent knowledge in a modular way. In our approach there is an initial macro-expansion phase during which macro calls are appropriately replaced by AnsProlog code. The result of the macro-expansion phase is an AnsProlog program which can then be used by an appropriate interpreter. In this paper we will use the Smodels [Niemelä & Simons 1997] interpreter for illustration purposes. The organization of the rest of the paper is as follows. We will first present a simple example of our approach, then we will present the syntax and semantics for our language constructs and then introduce a detailed illustration with respect to planning and reasoning about actions. Finally, we will conclude and discuss related work.

2 A simple example: transitive closure

Let us consider the simple example of transitive closure. We will illustrate how a simple transitive closure module can be defined once and can then be used in many different ways. A transitive closure module of a binary predicate p is computed by the binary predicate q, and is given as follows.

```
Module_name: Transitive_closure.
Parameters(Input: p(X,Y); Output: q(X,Y);). Types: Z = type X.
Body:  q(X,Y) :- p(X,Y).
       q(X,Y) :- p(X,Z), q(Z,Y).
```

Now, if in a program we want to say that anc [1] is the transitive closure of par [2] then we can have the following macro call in that program:

CallMacro Transitive_closure(**Replace**: *p* **by** *par*, *q* **by** *anc*, *X* **by** *U*, *Y* **by** *V*;).
Our semantics of the macro call will be defined in such a way that during the macro-expansion phase the above call will be replaced by the following rules, together with type information about the variable Z.

[1] $anc(a, b)$ means that b is an ancestor of a.
[2] $par(a, b)$ means b is a parent of a.

```
anc(U,V) :- par(U,V).
anc(U,V) :- par(U,Z), anc(Z,V).
```

Now suppose in another program we would like to define descendants of a, where $descendant(a, b)$ means that a is a descendant of b, then one can include one of the following macro calls:

CallMacro Transitive_closure(**Replace**: *p* **by** *par*, *q* **by** *descendant*, *Y* **by** *a*; **Unchanged**: X;).
CallMacro Transitive_closure(**Replace**: *p* **by** *par*, *q* **by** *descendant*; **Specialize**: *Y = a*; **Unchanged**: X;).

Our semantics of the macro call will be defined in such a way that during the macro-expansion phase the above calls will be replaced by the following rules and type information about Z.

```
descendant(X,a) :- par(X,a).
descendant(X,a) :- par(X,Z), descendant(Z,a).

descendant(X,Y) :- par(X,Y), Y = a.
descendant(X,Y) :- par(X,Z), descendant(Z,Y), Y = a.
```

A similar example is given in [McCarthy 1993]. There McCarthy gave a context in which above(x,y) is the transitive closure of on(x,y) and wrote lifting rules to connect this theory to a blocks world theory with on(x,y,s) and above(x,y,s).

3 Syntax and Semantics of modules and macro calls

We now present the syntax and semantics of modules and macro calls. We start with the alphabet. Our alphabet has module names, predicate names, variable names, function names and the following set of keywords '**Module_name**', '**Parameters**', '**#domain**', '**Input**', '**Output**', '**Types**', '**type**', '**Body**', '**Callmacro**', '**specializes**', '**generalizes**', '**variant_of**', '**Specialize**', '**Generalize**', '**Unchanged**', '**Replace**', '**by**', '**Add**', '**to**', '**Remove**' and '**from**'. We use the terminology of atoms, literals, naf-literals etc. from [Baral 2003]. Recall that naf-literal is either an atom or an atom preceded by the symbol '**not**'. Besides that, if p is a predicate of arity k, and $V_1, \ldots, V_k$ are terms, then we refer to $p(V_1, \ldots, V_k)$ as a predicate schema. Furthermore, we define a variable domain statement to be of the form $\#domain\ p(V)$, which says that the variable V is of type p. For example $\#domain\ fluent(F)$ means that the variable F is of the type $fluent$.

3.1 Syntax

We start with the syntax of a call-macro statement and then define a module.

Definition 1. A call-macro statement is of the following form:

$Callmacro\ Mname($ **Replace**: p_1 **by** $p'_1, \ldots, p_k$ **by** p'_k, v_1 **by** $v'_1, \ldots, v_l$ **by** v'_l; **Add**: u_1 **to** $q_1, \ldots, u_r$ **to** q_r; **Remove**: w_1 **from** $q'_1, \ldots, w_s$ **from** q'_s; **Specialize**: $S_1, \ldots, S_m$; **Generalize**: $G_1, \ldots, G_n$; **Unchanged**: $x_1, \ldots, x_t$;)

where $Mname$ is a module name, $p_1, \ldots, p_k$, $p'_1, \ldots, p'_k$, $q_1, \ldots, q_r$ and $q'_1, \ldots, q'_r$ are predicate names; $v_1, \ldots, v_l$, $v'_1, \ldots, v'_l$ are terms; $u_1, \ldots, u_r$ are sets of terms; $w_1, \ldots, w_s$ are sets of variables; x_1 to x_t are variables or predicates; S_is and G_js are naf-literals; $\{S_1, \ldots, S_m\} \cap \{G_1, \ldots, G_n\} = \emptyset$. Any of k, l, r, s, m, n or t could be 0. Also, the order in which we specify the keywords does not matter. □

Definition 2. A $module$ is of the form:

Module_Name: $Mname\ sg\ Mname'$.
Parameters($P_1 \ldots P_t$; Input: $I_1, \ldots, I_k$; Output: $O_1, \ldots, O_l$;).
Types: $D_0, \ldots D_j$, $L_1 = type\ V_1, \ldots L_o = type\ V_o$.
Body: $r_1 \quad \ldots \quad r_m$.
$\quad c_1 \quad \ldots \quad c_n$.

where, $Mname$, and $Mname'$ are module names; sg is either the keyword 'specializes', the keyword 'generalizes' or the keyword 'variant_of'; P_is, I_is and O_is are predicate schemas; r_is are AnsProlog rules (we also allow for Smodels constructs such as '#const' etc.); c_js are call-macro statements; $L_1, \ldots, L_o$ and $V_1, \ldots, V_o$ are variables; and $D_0, \ldots, D_j$ are variable domain statements. $Mname'$ is optional and in its absence we do not have the sg part.

But if $Mname'$ is there and sg is equal to 'specialize' or 'generalize', then $m = 0$, $n = 1$ and only sg appears in c_1. In other words, if sg is equal to specialize, then there is exactly one call to the module $Mname'$ using specialize and not generalize (similarly for generalize), and there are no other rules or macro calls. The idea of specifying specialize, generalize and variant between modules is to show the connection between the modules and if one is familiar with a module then it becomes easier for him/her to grasp the meaning of a specialization, generalization or variant of that module.

Additionally, we specify the parameters of the module, e.g. predicates and variables that are passed in and out from the module. We may define those in general, or further specify them to be input or output predicates or variables. The input and output labeling is optional, but are useful to express more information about the module. As shown in the upcoming examples, specifying inputs and outputs helps with understanding and usage of the particular module. In cases where $k = 0$ ($l = 0$) we may omit the $Input$ ($Output$) keyword. However, if input or output is present, we require the following conditions to hold:

(i) If p is an input predicate, then there must be a rule r_i whose body has p, or there must be a call-macro statement c_j with p in it.

(ii) If p is an output predicate, then there must be a rule r_i whose head has p, or there must be a call-macro statement c_j with p in it.

□

The above conditions ensure that the input and output predicates play their intended role in a module. Intuitively, a module takes in facts of a set of input predicates and reasons with them to produce a set of facts about output predicates. This is similar to the interpretation of logic programs as lp-functions in [Gelfond & Gabaldon 1997]. The first condition above requires that each of the specified inputs is actually used within

the module, while the second one ensures that the module really computes each of the specified outputs.

Let us now take a closer look at the variables in a module and their domains. First, we say a variable is *local*, if it does not appear in any parameter statement of the module. Otherwise, we say the variable is *global*. Our syntax allows for defining the domain of a variable either using '$\#domain$' statement, or by type constraints of the form $V = type\ V'$ meaning that the type of variable V is the same as the type of V'. We require that domains be only defined for local variables, as global variables get their domain from the macro calls. For example in the transitive closure module, defined in previous section, we do not want to specify the types of X and Y, as X and Y can be person, number, etc. The local variables must have a well-defined type, which is formally defined as follows:

A local variable V has a well-defined type if one of the following holds:

1. The definition of the module $Mname$ contains a statement $\#domain\ p(V)$.
2. The definition of the module $Mname$ contains a statement $V = type\ V'$ where V' is a global variable.
3. The definition of the module $Mname$ contains a statement $V = type\ V'$ where V' has a well-defined type.

In addition, we require the following condition to hold for any macro call:

(iii) If X is a predicate or variable in any parameter schema of $Mname$, then any macro call to the module $Mname$ must contain either a replace, remove, generalize or unchanged statement involving X. Furthermore, a macro call to a module can not refer to any local variable of that module. Finally, although we do not require it, it is advisable to make sure that any variable that is introduced (by the u_i notation of the Add statements) by a macro call to a module is either different from existing variable in that module or if same, has a reason behind it. (In our implementation we will flag such variables.)

3.2 Macro expansion semantics

To characterize the expansion of modules we need to consider not just a single module but a collection of modules, as a module may include call-macro statements that call other modules. Given a set of modules S we define the dependency graph G_S of the set as follows: There is an edge from M_1 to M_2 if the body of M_1 has a call-macro statement that calls M_2. In the following we only consider the sets of modules whose dependency graph does not have cycles.

Now given a set of modules its macro expansion semantics is a mapping λ from module names to AnsProlog programs. We define this mapping inductively as follows:

1. If M is a module with no macro calls then $\lambda(M) = \{r_1, \ldots, r_m\}$.
2. If c is a call-macro statement in module $Mname$ of the form

   $Callmacro\ Mname($ **Replace**: p_1 **by** $p'_1, \ldots, p_k$ **by** p'_k, v_1 **by** $v'_1, \ldots, v_l$ **by** v'_l; **Add**: u_1 **to** $q_1, \ldots, u_r$ **to** q_r; **Remove**: w_1 **from** $q'_1, \ldots, w_s$ **from** q'_s; **Specialize**: $S_1, \ldots, S_m$; **Generalize**: $G_1, \ldots, G_n$; **Unchanged**: $x_1, \ldots, x_t$;)
   such that $\lambda(M)$ is defined, then $\lambda(c)$ is defined as follows:

(a) Each rule r in $\lambda(M)$ is replaced by a rule r' constructed from r by applying all of the following(if applicable) changes to r:

i. (Replace) Let $p_i \in r$, $i = 1, ..., k$. Then p_i is replaced by it's respective predicate p'_i in r'. Similarly, any of the terms v_1 to v_l in any predicate $p \in r$ is replaced by it's respective term v'_1 to v'_l.

ii. (Add) Let $p(t_1, ..., t_i)$ be a predicate of r with it's respective terms. If for any j, $p = q_j$ and $u_j = \{t'_1, ..., t'_{i'}\}$, then $p(t_1, ..., t_i)$ is replaced by $p(t_1, ..., t_i, t'_1, ..., t'_{i'})$ in r'.

Example 1. Let p be the atom $q(Z, Y)$ of a rule r. Let the call contain the following: **Replace**: *q* **by** *occurs*; **Add**: {*A,neq(B)*} **to** *q*; Following the above cases, $q(Z, Y)$ will be replaced by $occurs(Z, Y, A, neg(B))$ in r'. □

iii. (Remove) Let w be any variable in term t_i from the set w_j for some j. Let q'_j be any predicate of the form $q'_j(t_1, ..., t_i, ...t_a)$ in r (notice that t_i may be equal to w). Then q'_j is replaced by $q'_j(t_1, ..., t_{i-1}, t_{i+1}...t_a)$ in r' assuming $t - 1 \geq 1$ and $t + q \leq a$ Otherwise the respective t_{i-1} or t_{i+1} is not present in p.

iv. (Unchanged) For any predicate or variable x_i, no change or substitution is performed.

v. If there exists i, j such that $p_i = p_j$ and $p'_i \neq p'_j$, or $v_i = v_j$ and $v'_i \neq v'_j$ then we say that the set of substitutions is conflicting. If that is the case, we say that the semantics of the call c, $\lambda(c)$ is undefined.

(b) (Specialize, Generalize) $S_1, ..., S_m$ is added to and $G_1, \ldots, G_n$, if present, are removed from the body of each of the rules of the module.

(c) (Local variables types) For each local variable L of $Mname$, it's type is assigned as follows. The type of L is assigned according to the $\#domain\ p(L)$ statement (if present in $Mname$) or type constraint $L = type\ V$ for some variable V with already defined type (i.e. a global or well-defined local variable). Notice that our syntax requires each local variable to be well-defined. Then, in the first case, the type of L is p, while in the latter case type of L is the same as type of V, where V is a global variable or a well-defined local variable.

(d) If $S_1, ..., S_m$ include evaluable predicates or equality predicates then appropriate simplification is done.

3. For a module M, such that $\lambda(c_1), \ldots, \lambda(c_n)$ are already defined $\lambda(M)$ is defined as follows:

$$\lambda(M) = \{r_1, \ldots, r_m\} \cup \lambda(c_1) \cup \ldots \cup \lambda(c_n)$$

Definition 3. Let S be a set of modules. Two modules M and M' are said to be rule-set equivalent (in S)[3] if $\lambda(M)$ and $\lambda(M')$ have the same set of rules modulo changes in the ordering of naf-literals in the body of a rule. □.

[3] When the context is clear we do not mention S explicitly.

4 Examples of simple specialization and generalization

In this section we illustrate some simple examples of specialization and generalization. We start with a simple module of inertial reasoning which says if F is true in the index T then it must be true in the index T'.

```
Module_Name: Inertia.
Parameters(Input: holds(F,T); Output: holds(F,T');).
Body:  holds(F,T') :- holds(F,T).
```

Consider the following call-macro statement.

CallMacro Inertia(**Replace**: *F* **by** *G*, *T'* **by** *res(A, T)*; **Unchanged**: holds, T;).
When the above call-macro statement is expanded we obtain the following:

```
holds(G,res(A,T)) :- holds(G,T).
```

Consider a different call-macro statement.

CallMacro Inertia(**Replace**: *F* **by** *G*, *T'* **by** *(T+1)*; **Unchanged**: holds, T;).
When the above call-macro statement is expanded we obtain the following:

```
holds(G,T+1) :- holds(G,T).
```

Now let us define some modules that specialize the module 'Inertia'.

(i) Inertia1

```
Module_Name: Inertia1 specializes Inertia.
Parameters(Input: holds(F,T); Output: holds(F,T');).
Body:  Callmacro Inertia(Unchanged: holds, F, T, T';
         Specialize: not ~holds(F,T'), not ab(F,T,T');).
```

Proposition 1. The module Inertia1 is rule-set equivalent to Inertia1' below. □.

```
Module_Name: Inertia1'.
Parameters(Input: holds(F,T), ab(F,T,T');Output: holds(F,T');).
Body: holds(F,T') :- holds(F,T), not ~holds(F,T'),
                     not ab(F,T,T').
```

(ii) Inertia2

```
Module_Name: Inertia2 variant_of Inertia.
Parameters(Input: holds(F,T); Output: holds(F,T');).
Body: Callmacro Inertia(Unchanged: holds, F, T, T';
        Specialize: not ~holds(F,T');).
      Callmacro Inertia(Replace: holds by ~holds;
        Unchanged: F, T, T'; Specialize: not holds(F,T');).
```

Note that in the above module we say 'variant_of' instead of 'specialize'. That is because the body of the above module has two macro calls and when using 'specialize' we only allow for one macro call.

Proposition 2. The module Inertia2 is rule-set equivalent to Inertia2' below. □.

```
Module_Name: Inertia2'.
Parameters(Input: holds(F,T); Output: holds(F,T');).
Body: holds(F,T')  :- holds(F,T), not ~holds(F,T').
      ~holds(F,T') :- ~holds(F,T),  not holds(F,T').
```

(iii) Inertia3

```
Module_Name: Inertia3 specializes Inertia.
Parameters(Input: holds(F,T); Output: holds(F,T');).
Body: Callmacro Inertia(Unchanged: holds, F, T, T';
                        Specialize: not ab(F,T,T');).
```

Proposition 3. The module Inertia3 is rule-set equivalent to Inertia3' below. □.

```
Module_Name: Inertia3'.
Parameters(Input: holds(F,T), ab(F,T,T');Output: holds(F,T');).
Body: holds(F,T') :- holds(F,T), not ab(F,T,T').
```

(iv) Inertia4

```
Module_Name: Inertia4 generalizes Inertia3.
Parameters(Input: holds(F,T); Output: holds(F,T');).
Body: Callmacro Inertia3(Unchanged: holds, F, T, T';
                         Generalize: not ab(F,T,T');).
```

Proposition 4. The module Inertia4 is rule-set equivalent to the module Inertia. □.

The above modules show how one can define new modules using previously defined modules by generalizing or specializing them. This is similar to class-subclass definitions used in object oriented programming languages. A specialization is analogous to a subclass while a generalization is analogous to a superclass. Now let us consider several call-macro statements involving the above modules.

(a) The statement "CallMacro Inertia2(**Replace**: *F* **by** *G*, *T* **by** *X*, *T'* **by** *X+1*; **Unchanged**: holds;)", when expanded [4], gives us the following rules:

```
holds(G,X+1)  :- holds(G,X), not ~holds(G,X+1).
~holds(G,X+1) :- ~holds(G,X), not holds(G,X+1).
```

(b) Similarly, the statement "CallMacro Inertia2(**Replace**: *F* **by** *G*, *T* **by** *X*, *T'* **by** *res(A,X)*; **Unchanged**: holds;)" when expanded will result in the following rules:

```
holds(G,res(A,X))  :- holds(G,X), not ~holds(G,res(A,X)).
~holds(G,res(A,X)) :- ~holds(G,X), not holds(G,res(A,X)).
```

The above illustrates how the same module Inertia2 can be used by different knowledge bases. *The first call-macro statement is appropriate to reason about inertia in a narrative while the second is appropriate to reason about inertia with respect to hypothetical situations.*

[4] All such statements in the rest of this paper can be thought of as formal results. But since their proofs are straight forward we refrain from adding a whole bunch of propositions.

5 Modules for planning and reasoning about actions

In this section we present several modules that we will later use in planning and reasoning about actions. In the process, we will show how certain modules can be used through appropriate macro calls in different ways.

5.1 Forall

We start with a module called 'Forall' defined as follows:

```
Module_Name: Forall.
Parameters(Input: in(X,S), p(X,T); Output: all(S,T);).
Body: ~all(S,T) :- in(X,S), not p(X,T).
      ~all(S,T) :- in(neg(X),S), not ~p(X,T).
      all(S,T)  :- not ~all(S,T).
```

Intuitively, the above module defines when all elements of S (positive or negative fluents) satisfy the property p at time point T. Now let us consider call-macro statements that call the above module.

• The statement "CallMacro Forall(**Replace**: *X* **by** *F*, *p* **by** *holds*, *all* **by** *holds_set*; **Unchanged**: S, T, in;)" when expanded will result in the following rules:

```
~holds_set(S,T) :- in(F,S), not holds(F,T).
~holds_set(S,T) :- in(neg(F),S), not ~holds(F,T).
holds_set(S,T)  :- not ~holds_set(S,T).
```

• The statement "CallMacro Forall(**Replace**: *X* **by** *F*, *in* **by** *finally*, *p* **by** *holds*;**Remove**: $\{S\}$ **from** *all*, $\{S\}$ **from** *in*, $\{S\}$ **from** *p*; **Unchanged**: all, T;)" when expanded will result in the following rule:

```
~all(T) :- finally(F), not holds(F,T).
~all(T) :- finally(neg(F)), not ~holds(F,T).
all(T)  :- not ~all(T).
```

The above rules define when all goal fluents (given by the predicate 'finally') are true at a time point T. Although the module specification of 'Forall' has an extra variable S, when the above macro call is expanded, S is removed.

5.2 Dynamic causal laws

Now let us consider a module that reasons about the effect of an action. The effect of an action is encoded using $causes(a, f, s)$, where a is an action, f is a fluent literal and s is a set of fluent literals. Intuitively, $causes(a, f, s)$ means that a will make f true in the 'next' situation if all literals in s hold in the situation where a is executed or a is to be executed.

```
Module_name: Dynamic1.
Parameters(Input: causes(A,F,S), holds_set(S,T);
           Output: holds(F,T');).
Body: holds(F,T')  :- causes(A,F,S), holds_set(S,T).
      ~holds(F,T') :- causes(A,neg(F),S), holds_set(S,T).
```

Now let us consider call-macro statements that call the above module.

• The statement "CallMacro Dynamic1(**Replace**: *F* **by** *G*, *T* **by** *X*, *T'* **by** *X+1*; **Specialize**: *occurs(A,X)*; **Unchanged**: A, S, holds, causes, holds_set;)" when expanded will result in the following rules:

```
holds(G,X+1)  :- occurs(A,X), causes(A,G,S), holds_set(S,X).
~holds(G,X+1) :- occurs(A,X), causes(A,neg(G),S),
                 holds_set(S,X).
```

• The statement "CallMacro Dynamic1(**Replace**: *F* **by** *G*, *T* **by** *X*, *T'* **by** *res(A,X)*; **Unchanged**: A, S, holds, causes, holds_set;)" when expanded will result in the following rules:

```
holds(G,res(A,X))  :- causes(A,G,S), holds_set(S,X).
~holds(G,res(A,X)) :- causes(A,neg(G),S),holds_set(S,X).
```

• The statement "CallMacro Dynamic1(**Replace**: *F* **by** *G*, *T* **by** *X*, *T'* **by** *X+D*; **Specialize**: *occurs(A,X)*, *duration(A,D)*; **Unchanged**: A, S, holds, causes, holds_set;)" when expanded will result in the following rules:

```
holds(G,X+D)  :- causes(A,G,S), holds_set(S,X),
                 occurs(A,X), duration(A,D).
~holds(G,X+D) :- causes(A,neg(G),S), holds_set(S,X),
                 occurs(A,X), duration(A,D).
```

The above illustrates how the module Dynamic1 can be used in three different ways: when reasoning about narratives where each action has a unit duration, when reasoning about hypothetical execution of actions, and when reasoning about narratives where each action has a duration that is given.

5.3 Enumeration

```
Module_Name: Enumerate1.
Parameters(Input: r(X), s(Y);Output: q(X,Y);). Types:Z=type X.
Body: ~q(X,Y) :- q(Z,Y), X!=Z, s(Y).
      q(X,Y)  :- r(X), s(Y), not ~q(X,Y).
```

The statement "CallMacro Enumerate1(**Replace**: *r* **by** *action*, *s* **by** *time*, *q* **by** *occurs*, *X* **by** *A*, *Y* **by** *T*;)" when expanded will result in the following rules:

```
~occurs(A,T) :- occurs(Z,T), A!=Z, time(T).
occurs(A,T)  :- action(A), time(T), not ~occurs(A,T).
```

5.4 Initialize

```
Module_Name: Initialize.
Parameters(Input: initially(F); Output: holds(F,0);).
Body: holds(F,0)  :- initially(F).
      ~holds(F,0) :- initially(neg(F)).
```

6 Planning

In this section we show how we can specify a planning program (and also a planning module) using call-macro statements to modules defined in the previous section.

6.1 An AnsProlog planning program in Smodels syntax

We start with a program that does planning. In the following program we have two actions a and b, and two fluents f and p. The action a makes f true if p is true when it is executed, while the action b makes p false if f is true when it is executed. Initially p is true and f is false and the goal is to make f true and p false.

```
initially(neg(f)). initially(p).     causes(a,f,s).
in(p,s).           set(s).           causes(b, neg(p), ss).
in(f,ss).          set(ss).          action(a).
action(b).         fluent(p).        fluent(f).
finally(f).        finally(neg(p)).  #const length = 1.
time(0..length).   #domain fluent(F).#domain set(S).
#domain action(A). #domain time(T).  #show holds(X,Y).
#show occurs(X,Y).

holds(F,0)      :- initially(F).
~holds(F,0)     :- initially(neg(F)).
holds(F, T+1)   :- holds(F,T), not ~holds(F,T+1).
~holds(F, T+1)  :- ~holds(F,T), not holds(F,T+1).
holds(F,T+1)    :- occurs(A,T), causes(A,F,S), holds_set(S,T).
~holds(F,T+1)   :- occurs(A,T),causes(A,neg(F),S),holds_set(S,T).
~holds_set(S,T) :- in(F,S), not holds(F,T).
~holds_set(S,T) :- in(neg(F),S), not ~holds(F,T).
holds_set(S,T)  :- not ~holds_set(S,T).
o_occurs(A,T)   :- occurs(Z,T), A!=Z, time(T).
occurs(A,T)     :- action(A), time(T), not o_occurs(A,T).
~allgoal        :- finally(F), not holds(F,length+1).
~allgoal        :- finally(neg(F)), not ~holds(F,length+1).
allgoal         :- not ~allgoal.
                :- not allgoal.
```

6.2 A planning module that calls several macros

We now define a planning module that has many call-macro statements calling macros defined in the previous section.

```
Module_name: Simple_Planning.
Parameters(Input: initially(F), causes(A,F,S), finally(F),
  in(F,S), action(A), length, holds(F, T), holds_set(S, T),
  time(T); Output: occurs(A,T), allgoal;).
Body: Callmacro Initialize(Unchanged: initially, holds, F;).
      Callmacro Inertia2(Replace: T' by T+1;
                         Unchanged: holds, F, T;).
      Callmacro Dynamic1(Replace: T' by T+1;
                         Specialize: occurs(A,X);
```

```
            Unchanged: causes, holds_set, holds, A, F, S, T;).
      Callmacro Forall(Replace: X by F, p by holds, all by holds_set;
                       Unchanged: in, S, T).
      Callmacro Enumerate1(Replace: X by A, Y by T, r by action,
                           s by time, q by occurs;).
      Callmacro Forall(Replace: X by F, in by finally, p by holds,
                       all by allgoal; Remove: {S, T} from all,
                       {S} from in, {T} from p; Add: {length+1} to p;).
:- not allgoal.
```

6.3 A planning program that calls the planning module

A planning program that calls the planning module in Section 6.2 and which when expanded results in the planning program in will consist of the declaration (first 9 lines) of the module in Section 6.1 and the following call:

```
Callmacro Simple_Planning(Unchanged: F, S, initially, causes,
  finally, in, action, length, holds, holds_set, time, occurs,
  allgoal;).
```

7 Ensembles and associated modules

So far in this paper we have focused on macros and macro expansions. To take the reuse and independent development of modules in an object-oriented manner further we propose that modules be grouped together under a "heading". This is analogous to object-oriented languages such as Java where methods that operate on the objects of a class are grouped under that class. In other words the "headings" in Java are class names under which methods are grouped.

Before we elaborate on what we propose as "headings" for our purpose here, we first consider some examples from Java. A typical class in Java (from Chapter 3 of [Horstman 2005]) is $BankAccount$. Associated with this class are the methods $deposit$, $withdraw$ and $getBalance$. A subclass of $BankAccount$ (from Chapter 13 of [Horstman 2005]) is the class $SavingsAccount$. In Java, in the $SavingsAccount$ class definition one only specifies new methods as it automatically inherits all methods from the $BankAccount$ class. An example of a new method for the class $SavingsAccount$ is $addInterest$.

The questions we would now like to address are: How are modules, as defined in this paper, organized? If they are grouped, how are they grouped and under what "headings"? If they are grouped, how do inheritance and polymorphism manifest themselves?

We propose that the modules be grouped under "headings". That allows one to locate a module more easily, compare modules that are similar, notice duplicate modules, etc. In regards to what "headings" we should use for grouping the modules, we notice that a module has predicates specified in its parameters and each positions of these predicates have an associate class. Thus we define a notion of an *ensemble* as a pair consisting of a set of classes S and a set of relation schemas R and propose to use ensembles as "headings" under which modules are grouped.

An example of an ensemble is a set $S = \{action, fluent, time\}$ and $R = \{initially(fluent\text{-}literals)$, $causes(action, fluent\text{-}literals, set\ of\ fluent\ literals)$, $finally(fluent\text{-}literals)\ \}$. Associated with each ensemble are a set of modules about those classes and relation schemas.

Similar to the notion of classes and sub-classes in Java we define the notion of sub-ensembles as follows. Let $E = (S, R)$ be an ensemble and $E' = (S', R')$ be another ensemble. We say E' is a sub-ensemble of E if there is a total one-to-one function f from S to S' such that for all class $c \in S$, $f(c)$ is a sub-class of c and $R \subseteq R'$.

By $F(S)$ we denote the subset of S given by $\{f(c) \mid c \in S\}$. Let us assume S and S' are of the same cardinality and $R = R'$. In that case E' basically has specialized subclasses for the various classes in E. Thus E' inherits the original modules (in the absence of overriding) that are in E and it may have special modules. For example S' may have the class $move\_actions$ which are a sub-class of $actions$ [Lifschitz & Ren 2006] and thus E' may have additional modules about such a sub-class of actions.

The above definition allows more relation schemas in E' than E. On the surface of it this may be counter-intuitive, but the intuition becomes clear when we assume $S = S'$. In that case E' has more relation schemas, so it can have more modules than in E. Thus E' can inherit all modules that correspond to E and can have more modules. In addition E' may have some module of the same name as E. In that case when one is in E' the module definition there overrides the module of the same name in E. E' can also have more classes than E and the intuition behind it is similar to the above and becomes clear when one assumes $S \subseteq S'$.

Macro calls can be used inside modules. When calling modules one then needs to specify the ensemble name from where the module comes from. This is analogous to class.methods calls in Java.

When developing a large knowledge base we will have an ensemble which will consist of a set of class names and a set of relation schemas. It will have its own modules. This ensemble will automatically inherit (when not overridden by its own modules) from various of its super-ensembles. One needs to deal with the case when an ensemble has say two super-ensembles each of which have a module of the same name.

For a knowledge base, exactly one of its module will be the "main" module. This is analogous to the "main" method in Java. This module may contain rules as well as macro calls to other modules that are defined or inherited. The set of rules of this module, after macro expansion will be the program that will be run to obtain answer sets or used to answer queries.

8 Conclusion, related work and software availability

In this paper we have introduced language constructs – syntax and semantics, that allows one to specify reusable modules for answer set programming. We illustrate our approach with respect to the planning example, and present several modules that can be called from a planning program. We also hint at how some of those modules, such as inertia, can be used by a program that does hypothetical reasoning about actions. In particular, while the statement "CallMacro Inertia2(**Replace**: *F* **by** *G*, *T* **by** *X*, *T'* **by** *X+1*;

Unchanged: holds;)" can be used in a planning program or a program that reasons with narratives the statement "CallMacro Inertia2(**Replace**: *F* **by** *G*, *T* **by** *X*, *T'* **by** *res(A,X)*; **Unchanged**: holds;)" can be used for hypothetical reasoning about actions. Note that both of them call the same module Inertia2. This is what we earlier referred to as reuse of code.

Among other works, our work is close to [Calimeri *et al.* 2004], [Gelfond 2006], [Lifschitz & Ren 2006]. In [Calimeri *et al.* 2004] 'templates' are used to quickly introduce new predefined constructs and to deal with compound data structures. The approach in [Gelfond 2006] is similar to us, and in [Lifschitz & Ren 2006] modular action theories are considered. Our use of "Replace" and the resulting simpler parameter matching – than in our earlier version of the paper in AAAI'06 spring symposium, is inspired by [Lifschitz & Ren 2006], which was also presented in the same symposium. Earlier, Chen et al. [Chen, Kifer, & Warren 1993] proposed the language of Hi-log that allows specification similar to our transitive closure modules. Other works in logic programming that discuss modularity include [Bugliesi *et al.* 1994,Eiter *et al.* 1997,Etalle & Gabbrielli 1996] and [Maher 1993].

Besides the above and CYC [Guha 1990] most recent efforts on resources for large scale knowledge base development and integration have focused on issues such as ontologies [Niles & Pease 2001], ontology languages [Dean *et al.* 2002], [Horrocks *et al.* 2003], [rul 2005], [Boley *et al.* 2004], [Grosof *et al.* 2003], and interchange formats [Genesereth & Fikes 1992,com]. Those issues are complementary to the issue we touch upon on this paper.

An initial implementation of an interface and interpreter of modules and macro calls is available at http://www.baral.us/modules/. As this is being written, we are still fine tuning the implementation.

9 Acknowledgements

We thank Michael Gelfond, Joohyung Lee, John McCarthy, Steve Maiorano, other participants of the 2006 AAAI spring symposium and anonymous reviewers for their suggestions. This work was supported by a DOT/ARDA contract and NSF grant 0412000.

References

[Baral 2003] Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

[Baselice, Bonatti, & Gelfond 2005] Baselice, S.; Bonatti, P.; and Gelfond, M. 2005. Towards an integration of answer set and constraint solving. In *Proc. of ICLP'05*.

[Boley *et al.* 2004] Boley, H.; Grosof, B.; Kifer, M.; Sintek, M.; Tabet, S.; and Wagner, G. 2004. Object-Oriented RuleML. http://www.ruleml.org/indoo/indoo.html.

[Bugliesi *et al.* 1994] Bugliesi, M.; Lamma, E.; Mello, P. 1994. Modularity in logic programming. In *Journal of logic programming*, Vol. 19-20, 443–502.

[Calimeri *et al.* 2004] Calimeri, F.; Ianni, G.; Ielpa, G.; Pietramala, A.; and Santoro, M. 2004. A system with template answer set programs. In *JELIA*, 693–697.

[Chen, Kifer, & Warren 1993] Chen, W.; Kifer, M.; and Warren, D. 1993. A foundation for higher-order logic programming. *Journal of Logic Programming* 15(3):187–230.

[com] Common Logic Standard. http://philebus.tamu.edu/cl/.

[Dean *et al.* 2002] Dean, M.; Connolly, D.; van Harmelen, F.; Hendler, J.; Horrocks, I.; McGuinness, D.; Patel-Schneider, P.; and Stein, L. 2002. OWL web ontology language 1.0 reference. http://www.w3.org/TR/owl-ref/.

[Eiter *et al.* 1997] Eiter, T.; Gottlob, G.; Veith, H. 1997. Modular logic programming and generalized quantifiers. In *Proc. 4th international conference on Logic programming and non-monotonic reasoning*, 290–309. Springer

[Etalle & Gabbrielli 1996] Etalle, S.; Gabbrielli M. 1996. Transformations of CLP modules. In *Theoretical computer science*, Vol. 166, 101–146.

[Gelfond & Gabaldon 1997] Gelfond, M., and Gabaldon, A. 1997. From functional specifications to logic programs. In Maluszynski, J., ed., *Proc. of International symposium on logic programming*, 355–370.

[Gelfond & Lifschitz 1988] Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R., and Bowen, K., eds., *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, 1070–1080. MIT Press.

[Gelfond 2006] Gelfond, M. 2006. Going places - notes on a modular development of knowledge about travel. In *Proceedings of AAAI 06 Spring Symposium: Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*.

[Genesereth & Fikes 1992] Genesereth, M., and Fikes, R. 1992. Knowledge interchange format. Technical Report Technical Report Logic-92-1, Stanford University.

[Grosof *et al.* 2003] Grosof, B.; Horrocks, I.; Volz, R.; and Decker, S. 2003. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proceedings of 12th International Conference on the World Wide Web (WWW-2003)*.

[Guha 1990] Guha, R. 1990. Micro-theories and Contexts in Cyc Part I: Basic Issues. Technical Report MCC Technical Report Number ACT-CYC-129-90.

[Horrocks *et al.* 2003] Horrocks, I.; Patel-Schneider, P.; Boley, H.; Tabet, S.; Grosof, B.; and Dean, M. 2003. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.daml.org/2003/11/swrl/.

[Horstman 2005] Horstman, C. 2005. *Big java*. John Wiley.

[Lifschitz & Ren 2006] Lifschitz, V. and Ren, W. 2006. Towards a Modular action description language. In *Proceedings of AAAI 06 Spring Symposium: Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*.

[Maher 1993] Maher, M. 1993. A transformation system for deductive databases modules with perfect model semantics. In *Theoretical computer science*, Vol. 110, 377–403.

[McCarthy 1993] McCarthy, J. 1993. Notes on formalizing contexts. In Bajcsy, R., ed., *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, 555–560. San Mateo, California: Morgan Kaufmann.

[Miller *et al.* 1990] Miller, G.; Beckwith, R.; Fellbaum, C.; Gross, D.; and Miller, K. 1990. Introduction to wordnet: An on-line lexical database. *International Journal of Lexicography (special issue)* 3(4):235– 312.

[Niemelä & Simons 1997] Niemelä, I., and Simons, P. 1997. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In Dix, J.; Furbach, U.; and Nerode, A., eds., *Proc. 4th international conference on Logic programming and non-monotonic reasoning*, 420–429. Springer.

[Niles & Pease 2001] Niles, I., and Pease, A. 2001. Towards a standard upper ontology. In *Proceedings of the international conference on Formal Ontology in Information Systems*, 2–9.

[rul 2005] 2005. RuleML: The Rule Markup Initiative. http://www.ruleml.org/.

[Tari, Baral, & Anwar 2005] Tari, L.; Baral, C.; and Anwar, S. 2005. A Language for Modular ASP: Application to ACC Tournament Scheduling. In *Proc. of ASP'05*.