

Macros, macro calls and use of ensembles in modular Answer Set Programming

Saadat Anwar, Chitta Baral and Juraj Dzifcak

Department of Computer Science and Engineering

Arizona State University

Tempe, AZ 85287

{saadat.anwar,chitta,juraj.dzifcak}

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. 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.

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) Macro: 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. Modules will correspond to the methods or subroutines in the libraries of other languages, such as Java or C/C++. In case of libraries, the actual library code is added to an application thus allowing the application to use methods or functions given by it.

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 the 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.

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 one time and can then be used in many different ways. A transitive closure module where p is binary predicate, whose transitive closure is computed by the binary predicate q , is given as follows.

```
Module name: Transitive_closure
Input: p(X,Y).      Output: q(X,Y).
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 is the transitive closure of par then we can have the following macro call in that program:

```
CallMacro Transitive_closure (input: par(U,V); output:
anc(U,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:

```
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 ancestors of a , then one can include one of the following macro calls:

```
CallMacro Transitive_closure( input=par(a,V) ;
output=anc(a,V)).
```

```
CallMacro Transitive_closure( input=par(U,V) ;
output=anc(U,V); specialize= U=a).
```

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:

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

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 $above(x,y,s)$.

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 special keywords ‘Callmacro’, ‘input’, ‘output’, ‘specialize’, ‘generalize’, and ‘variant of’. We use the terminology of atoms, literals, naf-literals etc. from (Baral 2003). Besides them if p is a predicate of arity k , and V_1, \dots, V_k are terms, then we refer to $p(V_1, \dots, V_k)$ as a predicate schema. We now define the syntax of a module.

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(input = I_1, \dots, I_k; output =
O_1, \dots, O_l; specialize = S_1, \dots, S_m; generalize =
G_1, \dots, G_n)
```

where $Mname$ is a module name, I_i s and O_j s are predicate schemas, S_i s and G_j s are naf-literals, and $\{S_1, \dots, S_m\} \cap \{G_1, \dots, G_n\} = \emptyset$. Any of k, l, m , and n could be 0. \square

Definition 2 A module is of the form:

```
Module Name: Mname sg Mname'
Input: I_1, \dots, I_k.      Output: O_1, \dots, O_l.
Body: r_1      \dots      r_m
      c_1      \dots      c_n
```

where, $Mname$, and $Mname'$ are module names, sg is either the keyword ‘specializes’ or the keyword ‘generalizes’ or the keyword ‘variant of’, I_i s and O_j s are predicate schemas, r_i s are AnsProlog rules and c_j s are call-macro statements. $Mname'$ is optional and in its absence we do not have the sg part.

But if $Mname'$ is there and sg is either ‘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 them 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.

Furthermore, 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 among the inputs.
- (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 among the outputs. \square

These conditions make sure 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 output. In the rest of the paper, we will often refer to a module of the above form by its name $Mname$.

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 $\lambda(M) = \{r_1, \dots, r_m\}$.
2. If c is a call-macro statement of the form *Callmacro* $M(\text{input} = I_1, \dots, I_k; \text{output} = O_1, \dots, O_l; \text{specialize} = S_1, \dots, S_m; \text{generalize} = G_1, \dots, G_n)$ such that $\lambda(M)$ is defined and M has input parameters I'_1, \dots, I'_k and output parameters O'_1, \dots, O'_l , then $\lambda(c)$ is defined as follows:
 - (a) Each rule r in $\lambda(M)$ is replaced by a rule r' constructed as follows:
 - i. For each atom a in r (either in the head or body of r) that has the same predicate as I'_i (or O'_j resp.), with θ as a mapping from (generic) positional variables to terms (that may or may not have that variable) such that applying that mapping to variables at various positions in I'_i we obtain a , we replace a by the atom a' obtained by using θ to map the variables at various positions of I_i (or O'_j resp.). The resulting substitution from variables in a to variables in a' , matched by their positions, is denoted by θ' .

Example 1 Let a be the atom $\text{in}(\text{neg}(X), S)$, I'_1 be the atom $\text{in}(X, S)$, and I_1 be $\text{in}(F, S)$. In this case θ maps X in the first position to the term $\text{neg}(X)$, and a S in the second position to itself. Thus it maps F to $\text{neg}(F)$ and S to S . Hence, a is replaced by $\text{in}(\text{neg}(F), S)$. The resulting substitution is then $\{X/F\}$. \square

Example 2 Let a be the atom $q(Z, Y)$, I'_1 be the atom $q(X, Y)$, and I_1 be $\text{occurs}(A, T)$. In this case θ maps X in the first position to the term Z and maps Y in the second position to itself. Thus it maps A to Z , and T to T . Hence, a is replaced by $\text{occurs}(Z, T)$. the resulting substitution is then $\{Y/T\}$. \square

- ii. Let $\theta_1, \dots, \theta_r$ be all such resulting substitutions. For all remaining atoms a in r (i.e., the ones which do not match with an input or output predicate I'_i or O'_j), a is replaced by the atom $a\theta_1\theta_2 \dots \theta_{r-1}\theta_r$.

Example 3 Let r be the rule:

$\text{o_q}(X, Y) :- \text{q}(Z, Y), \text{neg}(X, Z), \text{s}(Y)$.

Let $\text{o_q}(X, Y)$ be replaced by $\text{o_occurs}(A, T)$ with the resulting substitution $\{X/A, Y/T\}$; $\text{q}(Z, Y)$ be replaced by $\text{occurs}(Z, T)$ with the resulting

substitution $\{Y/T\}$; and $\text{s}(Y)$ be replaced by $\text{time}(T)$ with the resulting substitution $\{Y/T\}$. In that case the atom $\text{neg}(X, Z)$ is replaced by $\text{neg}(X, Z)\{X/A, Y/T\}\{Y/T\}\{Y/T\} = \text{neg}(A, Z)$. \square

- (b) S_1, \dots, S_m is added to and G_1, \dots, G_n , if present, are removed from the body of each of the rules of the program obtained in the previous step.
 - (c) If S_1, \dots, S_m include evaluable predicates or equality predicates then appropriate simplification is done.
3. For a module M , such that $\lambda(c_1), \dots, \lambda(c_n)$ are already defined $\lambda(M)$ is defined as follows:

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

Definition 3 Let S be a set of modules. Two modules M and M' are said to be equivalent (in S)¹ 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. \square

Note that two modules with the same set of rules but different set of input and output predicates are equivalent. However, notice that the calls to this modules are usually not, as the semantics of macro call depends on the input and output predicates given by the call.

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
Input: holds(F, T).
Output: holds(F, T').

Body: holds(F, T') :- holds(F, T).

Consider the following call-macro statement.

CallMacro Inertia(input=holds(G, T),
output=holds(G, res(A, 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(input=holds(G, T),
output=holds(G, T+1))

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

¹When the context is clear we do not mention S explicitly.

Module Name: Inertial specializes Inertia
 Input: holds(F, T).
 Output: holds(F, T').
 Body:
 Callmacro Inertia(input=holds(F,T);
 specialize = not ~holds(F,T'),
 not ab(F,T,T'); output=holds(F,T')).

Proposition 1 The module Inertial is equivalent to Inertial' below. \square

Module Name: Inertial'
 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

Input: holds(F,T).
 Output: holds(F,T').

Body:
 Callmacro Inertia(input=holds(F,T);
 specialize = not ~holds(F,T');
 output=holds(F,T')).
 Callmacro Inertia(input=~holds(F,T);
 specialize = not holds(F,T');
 output=~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 can have only one macro call.

Proposition 2 The module Inertia2 is equivalent to Inertia2' below. \square

Module Name: Inertia2'
 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
 Input: holds(F,T).
 Output: holds(F,T').

Body:
 Callmacro Inertia(input=holds(F,T);
 specialize = not ab(F,T,T');
 output=holds(F,T')).

Proposition 3 The module Inertia3 is equivalent to Inertia3' below. \square

Module Name: Inertia3'
 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
 Input: holds(F,T).
 Output: holds(F,T').

Body:
 Callmacro Inertia3(input=holds(F,T);
 generalize = not ab(F,T,T');
 output=holds(F,T')).

Proposition 4 The module Inertia4 is equivalent to the module Inertia. \square

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) "Callmacro Inertia2 (input = holds(G,X); output = holds(G,X+1))," when expanded ², 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 (input = holds(G,X); output = holds(G,res(A,X)))" 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.*

Modules that can be used in 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,

²All 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.

we will show how certain modules can be used through appropriate macro calls in different ways.

Forall

We start with a module called ‘forall’ defined as follows:

```
Module Name: forall
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 (input = in(F,S), holds(F,T), output = holds_set(S,T))” when expanded will result in the following rule:

```
holds_set(S,T) :- not ~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).
```

- The statement “Callmacro forall (input = finally(F), holds(F,T); output = 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 correctly ignored.

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.

```
Modulename: Dynamic1
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 (input = causes(A,G,S), holds_set(S,X); output = holds(G,X+1); specialize: occurs(A,X))” 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 (input = causes(A,G,S), holds_set(S,X); output = holds(G,res(A,X)))” 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 (input = causes(A,G,S), holds_set(S,X); specialize: occurs(A,X), duration(A,D); output = holds(G,X+D))” 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 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.

Enumeration

```
Module Name: enumerate1
Input: r(X), s(Y).
Output: q(X,Y).
```

```
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 (input = action(A), time(T); output = occurs(A,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).
```

Initialize

```
Module Name: initialize
Input: initially(F).
Output: holds(F,0).

Body:
holds(F,0) :- initially(F).
~holds(F,0) :- initially(neg(F)).
```

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.

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).
#hide.              #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.
```

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
Input: initially(F), causes(A,F,S),
       finally(F), in(F,S), action(A),
       length.
Output: occurs(A,T)
Body:
Callmacro initialize(
        input=initially(F);
        output=holds(F,0)).
Callmacro Inertia2(
        input=holds(F,T);
        output=holds(F,T+1)).
Callmacro Dynamic1(
        input=causes(A,F,S),
        holds_set(S,T);
        output=holds(F,T+1);
        specialize=occurs(A,X)).
Callmacro forall(
        input=in(F,S), holds(F,T);
        output=holds_set(S,T)).
Callmacro enumeratel(
        input=action(A), time(T);
        output=occurs(A,T)).
Callmacro forall(
        input=finally(F),
        holds(F,N);
        output=allgoal;
        specialize=eq(N, length+1)).
```

A planning program that calls the planning module

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

```
Callmacro Simple_Planning(
        input= initially(F),
        causes(A,F,S),
        length, finally(F),
        in(F,S), action(A);
        output=occurs(A,T)).
:- not allgoal.
```

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 sub-class 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 use to group the modules, we notice that a module has Input and Output predicates and each parameter 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 $R = \{initially(fluent-literals), causes(action, fluent-literals, set\ of\ fluent\ literals), finally(fluent-literals)\}$. 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* 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. The macro expansion may introduce new variables that are not in the Input or Output. The domain of these variables are inferred from the domain of other variables that appear in the same predicate at the same location.

Conclusion and related work

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 (input = holds(G,X); output = holds(G,X+1))
```

can be used in a planning program or a program that reasons with narratives, the statement

```
Callmacro Inertia2 (input = holds(G,X); output = holds(G,res(A,X)))
```

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.

Similarly, we show how an initial Inertia module can be specialized for different circumstances. This is analogous to the definition of classes and sub-classes in object-oriented programming, where the sub-classes normally inherit the methods of their super-class. In our terminology, a sub-class corresponds to a specialization of a module. In particular, while the statement

```
Callmacro Inertia2 (input = holds(G,X); output = holds(G,X+1))
```

can be used in a planning program or a program that reasons with narratives, the statement

```
Callmacro Inertia2 (input = holds(G,X); output = holds(G,res(A,X)))
```

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.

Similarly, we show how an initial Inertia module can be specialized for different circumstances. This is analogous to the definition of classes and sub-classes in object-oriented programming, where the sub-classes normally inherit the methods of their super-class. In our terminology, a sub-class corresponds to a specialization of a module.

Among other works, our work is close to (Calimeri *et al.* 2004); but their focus seem to be on aggregates and it is not clear if their language can express generalization and specialization that we have, and if it can express the various modules that we have presented in this paper. Earlier, Chen *et al.* (Chen, Kifer, & Warren 1993) proposed the language of Hi-log that allows specification similar to our transitive closure modules. 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; Grosf *et al.* 2003), and interchange formats (Genesereth & Fikes 1992; com). Those issues are complementary to the issue we touch upon on this paper.

References

- Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.
- Baselice, S.; Bonatti, P.; and Gelfond, M. 2005. Towards an integration of answer set and constraint solving. In *Proc. of ICLP'05*.
- Boley, H.; Grosf, B.; Kifer, M.; Sintek, M.; Tabet, S.; and Wagner, G. 2004. Object-Oriented RuleML. <http://www.ruleml.org/indoo/indoo.html>.
- Calimeri, F.; Ianni, G.; Ielpa, G.; Pietramala, A.; and Santoro, M. 2004. A system with template answer set programs. In *JELIA*, 693–697.
- Chen, W.; Kifer, M.; and Warren, D. 1993. A foundation for higher-order logic programming. *Journal of Logic Programming* 15(3):187–230.
- Common Logic Standard. <http://philebus.tamu.edu/cl/>.

- 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/>.
- 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, 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.
- Genesereth, M., and Fikes, R. 1992. Knowledge interchange format. Technical Report Technical Report Logic-92-1, Stanford University.
- Grosf, 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, R. 1990. Micro-theories and Contexts in Cyc Part I: Basic Issues. Technical Report MCC Technical Report Number ACT-CYC-129-90.
- Horrocks, I.; Patel-Schneider, P.; Boley, H.; Tabet, S.; Grosf, B.; and Dean, M. 2003. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. <http://www.daml.org/2003/11/swrl/>.
- Horstman, C. 2005. *Big java*. John Wiley.
- 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, 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, 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, I., and Pease, A. 2001. Towards a standard upper ontology. In *Proceedings of the international conference on Formal Ontology in Information Systems*, 2–9.
2005. RuleML: The Rule Markup Initiative. <http://www.ruleml.org/>.
- Tari, L.; Baral, C.; and Anwar, S. 2005. A Language for Modular ASP: Application to ACC Tournament Scheduling. In *Proc. of ASP'05*.