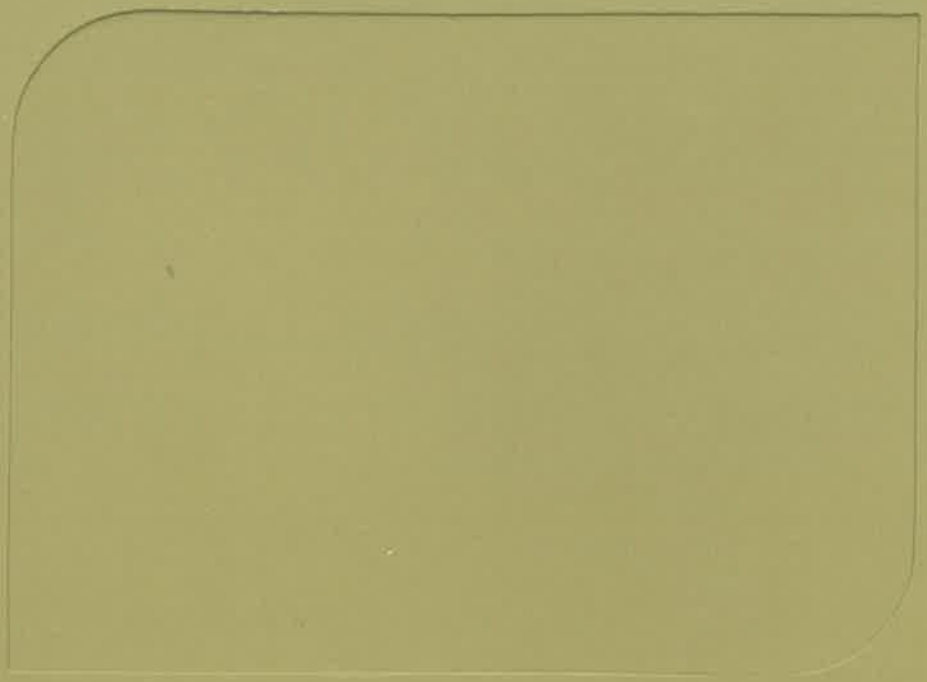

# BIM-PROLOG

Joint Project between
BIM
and
Department of Computer Science
Katholieke Universiteit LEUVEN

Sponsored by DPWB/SPPS
under grant nr KBAR/SOFT/1

# Documentatie over modules in BIM-prolog

by

Bart DEMOEN *

Technical Memorandum
BIM-prolog TM9

June 1985

* BIM
Kwikstraat 4
B-3078 Everberg Belgium
tel. +32 2 759 59 25

** Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A
B-3030 Heverlee Belgium
tel. +32 16 20 06 56

DPWB = Diensten van de eerste minister : Programmatie van het Wetenschapsbeleid.
SPPS = Services du premier ministre : Programmation de la Politique Scientifique.

# DOCUMENTATION ON MODULES IN BIM-PROLOG

DEMOEN Bart

BIM
Kwikstraat 4
B-3078 Everberg

Report
BIM-prolog TM9
June 1985

This paper concerns the nature and use of BIMprolog modules. Initially an overview is given, followed by details, such as a discussion of the interaction between modules and other directives concerned with operators, dynamic code, data bases etc. The influence of modules on certain built-in predicates is also discussed. The discussion is illustrated with examples.

Note that it is quite possible to program without modules, and that programs already written without the use of modules will remain executable without adjustment.

## 1 Overview

This overview discusses using modules, only within the context of BIMprolog programs (stored on file), which are to be compiled and consulted. The use of modules in an interactive session is discussed later.

Modules are to be used as a tool in the development of large BIMprolog programs.

A module is a set of clauses, to which a module name is given. This name can be given explicitly to a module, using the directive :

```
:- module(name) .
```

A module to which no explicit name has been given has the empty string as its name. A module without a name is called the global module, for reasons which will appear later. This directive can only appear once in a file.

Definitions within a module are usually local to that module, i.e. the

predicate which is defined cannot be called from another module. However, this default can be overruled using the directives :

```
:- import a/3 from module1 .
```

or

```
:- global b/2 .
```

The first directive occurs in a module (e.g. module2) in which you want to use a/3 from module1, and causes each call from module2 of a/3, to be interpreted as a call from the predicate with name a, arity 3, and defined in module1.

The second directive can be written in the module containing the definition of the predicate b/2, and has the effect that each module can call b/2, without having to use an explicit import declaration for b/2.
It is as though a global declaration is given, for all the built-in predicates. In fact, builtins written in BIMprolog are defined in a module with name system and are made global.

Definitions inside a module without a name are global. Thus a module without a name is called the global module.

In order to avoid such a global declaration, (e.g. by redefining a built-in predicate, local to a module), the following declaration can be used :

```
:- local write/1 .
```

which permits definitions of write/1 to be written in the (non global) module. The presence of write/1, then refers to the local definition.

The directive :

```
:- export c/4 .
```

is used to make the predicate c/4 available for modules which want to import it. The export directive also allows checking whether all imported predicates were intended to be imported. However, these checks have not yet been implemented, so that an export declaration is treated as a local declaration.

Thus far, discussion has been restricted to predicates, i.e. functors for which a definition exists, or for which a call exists. The above is also true for other functors and atoms, an atom being simply a functor of arity Ø. In this way, not only can procedures be local, global, imported or exported, but data structures as well.

This encourages the use of "information hiding", which is associated with the use of modules. Each functor thus has a module qualification.

There is, however, an important difference between the defaults for

predicates with a definition and for functors without a definition. This will be discussed in detail later. Simple rules exist which can be used to decide to which module a functor belongs.

It is possible to import the same functor, from 2 different modules. e.g. import a/3 from module1, and from module2 :

```
Declare -
        :- import a/3 from module1 .
        :- import a/3 from module2 .
```

in which case, a call such as a(_x,[],_y) is ambiguous. This ambiguity can be removed by using syntactic sugar.

If a/3 from module1 is intended, write -

```
a$module1(_x,[],_y)
```

and correspondingly for module2. The syntactic sugar may only be used in the case of ambiguity. Any other use results in a compiler error. The syntactic sugar gives an explicit qualification to a functor.

Syntactic sugar extends the syntax given in section 3.
Note, there should be no space between the name of the functor and the $, nor between the $ and the name of the module.
%$/ is an atom without qualification, equivalent to '%$/'
To denote % with module qualification / , you must write '%'$'/'

## 2 Defaults

The following method can be used to decide which module a functor belongs to. This is done initially in the case of non ambiguity.

If a/3 is found in module1, without explicit module qualification, it belongs to :

the global module, if a directive :- global a/3 exists at

module1 if a directive :- local a/3 exists

module2 if a directive :- import a/3 from module2 exists

module1 if module1 contains a procedure definition of a/3

the global module in the other cases

In the case of ambiguity, with explicit module qualification, the explicit qualification is accepted.

In the case of ambiguity, and without explicit module qualification, the functor is considered local to the module if there is a sensible local interpretation, e.g. there is a definition or a dynamic declaration ... )

In all other cases, the module conventions have been violated, and the compiler willemit the appropriate error message.

## 3 Interactive Mode

In interactive mode, more freedom is given than when a file is to be compiled. For example, explicit module qualification is allowed at all times. When a BIMprolog session starts (in the global module), none of the import, export, local or global declarations appearing in the consulted files have effect. If a predicate is loaded with name a and arity 4 and local to module modA, it will not be callable , unless the syntactic sugar is used or unless one is positioned within modA (using module/1).

Positioning within module modA is done with the query:

```
?- module(modA) .
```

The predicate a/4 from modA is now accessible without explicit qualification. Typing in new definitions, will cause them to be added to the current module. See the script of a session in the examples below.

## 4 Interaction with other directives

The directives op, dynamic, alldynamic, extern etc. influence the module qualification of functors.

```
:- extern(address,3,testdb) .
```

causes address/3 to be interpreted as a global functor.

:- op(100,fx,x) . defines the functor x/1 global provided that no other declarations for x/1 were given previously.

If the same declaration is preceded by :- module(modA) .

and :- local(x/1) .

then a local operator has been defined.

If the declaration was preceded by

```
:- import x/1 from modB .
```

then the functor x/1 from modB, has been made an operator in the file.

The dynamic declaration is only useful for predicates which are defined in the same module. That is, a dynamic declaration can apply to a global and a local predicate. For example :

```
:- module(one) .
:- dynamic a/4 .
:- global(b/2) .
:- dynamic b/2 .
```

make a/4 a local dynamic predicate, and b/2 a global dynamic predicate.

In the case of ambiguity, the explicit qualification should be used. For example :

```
:- module(one) .
:- local(a/3) .
:- import a/3 from two .
:- dynamic a/3  . {There is a possible local interpretation for a/3,
                          so there is no need for the explicit qualification .}
:- dynamic a$two/3  .
:- op(100,fx,a$one) .

{2 definitions of the local a/3}

a(_x,_y,_z) :- write(f(_x,_y,_z)) , a$two(_x,_y,_z) , ! .

a(_x,_x,_y) :- a(_y,_y,_x) . {a call of the local a/3}.
```

In this way, it is perfectly possible to write programs that are difficult to understand. Be warned!

```
:- alldynamic .
```

This causes every procedure in the file to be considered as a dynamic procedure. However, since the name/arity of the following definitions is not explicitly mentioned in this declaration, it is not equivalent to a dynamic declaration of all procedures defined in the module.

Compare

```
:- module(one) .
:- alldynamic .
:- import a/1 from two .

proc :- a(hello) .        {Refers to a/1 from module two.}
a(_x) :- write(_x) .      {Local dynamic definition of a/1}
```

with

```
:- module(one) .
:- dynamic a/1  .
:- import a/1 from two .

proc :- a(hello) .        {Refers to a/1 from module one}
a(_x) :- write(_x) .      {Local dynamic definition of a/1}
```

## 5 Extra built-ins

**module/1**

arg1 : in or output : Current module. (Must be an atom)

If instantiated, the current module becomes arg1.
If free, arg1 will be instantiated to the name of the current module.(See examples)

**module/2**

arg1 : input : Term.

arg2 : input or output : Module name.

Unify arg2 with the module qualification name of the principal functor of arg1. (arg1 may not be a free variable nor a number).

Example: ?- module(a$one(_x),_y) , write(_y) .

outputs one

**module/3**

arg1 : in or output : Term.

arg2 : input or output : Module name.

arg3 : output : Term.

arg3 is the term constructed from arg1 by stripping the

module qualification from the principal functor of arg1, and unifying this qualification with arg2.
If arg1 is free, arg2 must be an atom and arg3 must be partially instantiated.

Example: ?- module(a$one(b$two),one,a(b$two)) .

succeeds, if inside the global module.

mod_unif/2

arg1 : input : Term.

arg2 : input : Term.

Unify the 2 arguments, as if they had no module qualification.

Example:

```
?- mod_unif(a$one(_x),a$two(b$three)) ,
       module(_x,_y) , write(_x) , nl , writeq(_y) .
```

outputs: b
''

mlisting/1

arg1 : input : Module name. (Must be an atom).

Write all predicate definitions, that are local to the module whose name is arg1, on the current output stream.

## 6 Interaction with other built-in predicates

functor/3

Output arguments are always global.

all_directives/0
all_directives/1

Declarations concerning modules are not output.

readc/0
readc/1
read/0
read/1

Read global characters and terms.

write/1
write/2
writeq/1
writeq/2
display/1

| | |
|---|---|
| display/2 | |
| listing/0 | |
| listing/1 | |
| flisting/1 | |
| flisting/2 | |
| mlisting/1 | Write terms without qualification. (In a later release, output predicates that write with qualifications will be added) |
| is/2 | |
| ascii/2 | |
| atomtolist/2 | |
| name/2 | All atoms that are created by these predicates are global. The qualification of the instantiated atoms at the moment of the call, does not influence the working of the predicates. |
| assert/1 | |
| assert/2 | |
| asserta/1 | |
| assertz/1 | |
| clause/2 | |
| clause/3 | |
| retract/1 | |
| retract/2 | |
| retractall/1 | The qualification of the terms is important. |
| numbervars/3 | The variables are instantiated to global atoms. |

## 7 Examples

In the interactive sessions below, a number has been added to each prompt, to simplify explanation.

### Example 1

```
Script started on Wed Jun 19 17:29:46 1985
iris% BIMprolog
BIMMODULE_Prolog  - release 0.7 5-6-1985

1> ?- module(_x) , writeq(_x) , nl .

2> a$modA(_x) :- write('call of a$modA/1  ') , write(_x) .

3> a :- write('call of a global a/0  ') , write(_x) .
```

```
4> ?- a .
call of a global a/Ø   _Ø
5> ?- a$modA(hello) .
call of a$modA/1  hello
6> ?- a(hello) .
*** RUNTIME 22Ø *** Illegal call : unknown procedure a/1.

7> ?- module(modA) .

8> ?- module(_x) , writeq(_x) , nl .
modA

9> ?- a(hello) .
call of a$modA/1  hello
1Ø> ?- a .
call of a global a/Ø   _Ø
11> a(_x) :- write('definition added to a$modA/1') .

12> ?- a('hello ') .
call of a$modA/1  hello definition added to a$modA/1
13> b :- write(pok) .

14> ?- b.
pok
15> ?- module('') .

16> ?- b .
*** RUNTIME 22Ø *** Illegal call : unknown procedure b/Ø.

17> ?- mlisting(modA) .

a( _x ) :-
   write('call of a$modA/1 '),
   write( _x ).
a( _ ) :-
   write('definition added to a$modA/1').

b :-
   write(pok).


18> ?- stop .
script done on Wed Jun 19 17:33:13 1985
```

1: Start of the session. The current module is '' i.e. the global module.

2: A definition of a/1 belonging to modA.

3: A global definition of a/Ø

6: a/1 has no possible global interpretation, resulting in a warning.

7: Set the current module to modA.

8: Check whether the module is really modA.

9: a/1 has a local interpretation. It is executed.

10: a/0 has no local interpretation. It does have a global interpretation, so the global is called.

11: The definition is added to a$modA/1

13: The first definition of b$modA/0

15: Reset the current module to global.

16: b/0 is not known to the global module.

17: Print the predicates belonging to module modA.

Example 2
---

A listing of a BIMprolog source file is given, then a short session in which the source file is consulted and used.
Note the warning at line 2. Without the declaration at line 2, a redefinition of writeq/1 would be forbidden.

Listing for file mod1.pro

```
   1    :- module(one) .
   2    :- local writeq/1 .
*** WARNING  15 *** Local redefinition of a builtin predicate
   3    :- global ( test / 1) .
   4    :- local yes/0 .
   5
   6    writeq(_x) :- write(/*) , write(_x) , write(*/) , nl .
   7
   8    test(good) :- read(_y) , mod_unif(_y,yes) , ! .
   9    test(bad) .
  10
  11    loctest(good) :- read(_y) , _y = yes , ! .
  12    loctest(bad) .

Total number of errors      0
Total number of warnings    1
```

```
Compilation completed succesfully



Script started on Wed Jun 19 22:03:21 1985
iris% BIMprolog modl.pro
BIMprolog  - release 0.7 5-6-1985
modl.pro consulted.

1> ?- writeq(blabla) .
blabla
2> ?- test(_x) , write(_x) .

@ yes .
good
3> ?- loctest(_x) , write(_x) .
*** RUNTIME 220 *** Illegal call : unknown procedure loctest/1.

4> ?- module(one) .

5> ?- writeq(blabla) .
/*blabla*/

6> ?- test(_x) , write(_x) .

@ yes .
good
7> ?- loctest(_x) , write(_x) .

@ yes .
bad
8> ?- stop .
iris% ^
script done on Wed Jun 19 22:05:36 1985
```

1: writeq/1 is the (global) built-in predicate.

2: yes/0 is local in module one. read/1 reads global terms. mod_unif unifies the answer yes with yes$one

3: Without qualification, loctest/1 is inaccessible from the global module.

4: Set current module to one.

5: This time, writeq/1 refers to the local definition in module one.

6: Since there is no local definition of test/1, (being now in module one), a global definition is attempted.

7: The global atom yes, which is read by read/1, does not unify with the local atom yes/Ø of module one.

Example 3
---

The following example shows how to import a definition. It also shows that the module declaration need not be the first sentence in a source file. Anything occurring before the module declaration is considered global.

Listing for file rev.pro

```
 1    genlist(Ø,[]) :- ! .
 2    genlist(_n,[_n | _tail]) :- _m is _n - 1 , genlist(_m , _tail) .
 3
 4    :- module(reverse) .
 5    :- import append/3 from append .
 6    :- global(show/1) .
 7
 8    show(_n) :- genlist(_n,_list) , reverse(_list,_revlist) , write(_revlist) .
 9
10    reverse([],[]) .
11    reverse([_a | _rest] , _list) :- reverse(_rest,_revrest) ,
12                                     append(_revrest,[_a],_list) .
```

```
Total number of errors      Ø
Total number of warnings    Ø

Compilation completed succesfully
```

Listing for file app.pro

```
 1    :- module(append) .
 2
 3    append([],_l,_l) .
 4    append([_a | _l1] , _l2 , [_a | _l3]) :- append(_l1 , _l2 , _l3) .
```

```
Total number of errors      Ø
Total number of warnings    Ø

Compilation completed succesfully
```

```
Script started on Wed Jun 19 22:25:Ø1 1985
iris% BIMprolog rev.pro app.pro
BIMprolog  - release Ø.7 5-6-1985
rev.pro consulted.
app.pro consulted.

> ?- show(1Ø) .
[1,2,3,4,5,6,7,8,9,1Ø]
> ?- mlisting(append) .

append(nil, _1 , _1 ).
append([ _a | _11 ], _12 ,[ _a | _13 ]) :-
   append( _11 , _12 , _13 ).

> ?- mlisting(reverse) .

reverse(nil,nil).
reverse([ _a | _rest ], _list ) :-
   reverse( _rest , _revrest ),
   append( _revrest ,[ _a ], _list ).

> ?- mlisting('') .

genlist(Ø,nil) :-
   !.
genlist( _n ,[ _n | _tail ]) :-
    _m is _n - 1,
   genlist( _m , _tail ).

show( _n ) :-
   genlist( _n , _list ),
   reverse( _list , _revlist ),
   write( _revlist ).

> ?- stop .
iris% ^
script done on Wed Jun 19 22:25:58 1985
```