A Hierarchical Module Structure for WUP Programs

Chris Baird

Research Report CS-85-49
December 1985

ABSTRACT

Modular programming, decomposing programs into smaller components or modules, is a software engineering technique for managing the complexity of large software projects. Several imperative programming languages, including Modula-2 and Ada, have incorporated the concept of the module in their design. Logic programming languages, if they are to be used for writing large programs, must also provide features for modular programming.

The work reported here was intended to provide a new system of organizing Waterloo UNIX† Prolog **(WUP)** modules which would eliminate existing shortcomings. This paper describes a hierarchical module structure for WUP that is statically derived from the UNIX file structure. The module structure of a program may also be dynamically configured by the programmer. This implementation of modules in **WUP** is compared to that in other logic programming systems.

† UNIX is a Trademark of Bell Laboratories

Table of Contents

1. Introduction 3
2. Modular Programming 3
3. Modules in WUP 4
   - 3.1. WUP 1.4 vs. WUP 2.0 4
   - 3.2. Compiling 7
   - 3.3. Dynamic Structuring of Modules 10
   - 3.4. Searching for a Predicate 10
   - 3.5. Special Files 11
   - 3.6. Built-in Predicates 13
   - 3.7. Comparison 14
4. Using Modules in WUP 15
5. Modules in other Logic Programming Systems 17
   - 5.1. M-PROLOG 17
   - 5.2. micro-PROLOG 18
   - 5.3. HIMIKO 19
   - 5.4. Comparisons with WUP 20
6. Conclusion 23

References 25

Appendix A - Search Algorithm 26

1. Introduction

The value of modularity in software has long been recognized. Decomposing software into modules allows large programs to be organized in such a way as to be both manageable and intelligible. Monolithic software, a large single module program, is more difficult to write and to understand once written. Constructs for creating modules have been implemented in several imperative programming languages such as CLU [8], Modula [14], and most recently ADA [1]. If logic programming languages are to be used for large software projects it is essential that they provide features which support modularity. **WUP** (Waterloo UNIX Prolog) has, therefore, incorporated the concept of modular programming into a Prolog programming system.

2. Modular Programming

The basic element of modular program design is the abstraction. Abstraction is the taking away or leaving out of details. An abstraction is a definition of some desired behaviour; a module is the realization of that behaviour in a program. The user of a module is interested only in the abstract concept it represents rather than the details of the representation. Dealing with abstractions in this way makes it possible to manage complexity in software development.

Ghezzi and Jazayeri [7] define a number of requirements of the development process and development tools used for building large systems.

- the project must be divided among several people, each working independently
- the system should be built from modules, each independently designed and implemented, that are re-usable in other contexts
- the system must be easily modifiable - changes to one module should not effect other modules
- it must be possible to show correctness of the system based on the correctness of the individual modules

They also identify several important characteristics of the construction and use of modules which support the above objectives. Each module should correspond to either a procedural abstraction, i.e., a mapping from a set of input data to a set of output

data, or a data abstraction, i.e., an implementation of a data type comprising the specification of a set of data objects and the operations which manipulate them. These operations should be the only means of accessing the data objects. Therefore, a module interface should explicitly state which entities are exported from and imported to the module. Restricting knowledge about the implementation of an abstraction, or information hiding, by means of well defined interfaces simplifies the interactions and enhances the independence of program modules.

To preserve the concept of abstraction module independence is a necessity. The success of software projects depends, in large part, on this independence. Lack of module independence, leading to subtle interactions between modules, can create problems in all phases of software development from design and implementation, through testing and maintenance. The increase in complexity of software, resulting from module interaction, can make it unmanageable.

Modular programming techniques should serve the same purpose in logic programming as in von Neumann programming and provide the same benefits. It seems reasonable that, as a first step towards modular programming in Prolog, the constructs for implementing abstractions can be simply adopted from imperative languages. Logic programming is, however, different than von Neumann programming. Whether or not the practice of modular programming should or needs to be copied in total without alteration will be discovered only through experience.

3. Modules in WUP

3.1. WUP 1.4 vs. WUP 2.0

In the previous version of WUP a UNIX directory was considered to be a module and could contain any number of files and sub-directories. Clauses for different predicates were kept in separate files and the file name was required to be the same as the predicate name. Source files could be kept in any number of sub-directories to any

depth under the main directory. These files could be compiled separately or all at once by using the name of the main directory. The resulting **pure code** was stored in files under a single sub-directory **.db** of the main directory. For each source file, a sub-directory, with the same name, would be created under directory **.db.** Each sub-directory would, in turn, contain separate pure code files for all predicate with the corresponding name, each file having the arity of the predicate as its name. Figure 3.1 illustrates the structure of a WUP 1.4 module.

This system allowed easy retrieval of the pure code by simply building the UNIX file pathname from the predicate name and its arity. However it was extremely costly in terms of storage space and cumbersome in terms of the number of separate files that had to be maintained. There was no mechanism for statically organizing individual modules into larger program units. A module was included in a program by importing it, which simply added the module to the end of a list of previously imported modules. The organizational information that programmers often provided by distributing source files among several sub-directories of a module directory was lost when the program was compiled and collapsed into one pure code unit. The work reported here was intended to provide a new system of organizing WUP modules which would eliminate these shortcomings.

The new system provides a more flexible and more general implementation of the WUP module concept by using the UNIX hierarchical file structure to build an equivalent module structure.

Wirth [14] states that the essence of programming is finding the right, or at least an appropriate, structure for the program. A hierarchical structure is an effective means of program organization reflecting the top-down design techniques of structured programming methodology. Top-down design is the hierarchical analysis of a problem by breaking it into smaller parts, each conceptually subordinate to the one above it. The principle being applied is that of abstraction. At each level certain tasks are

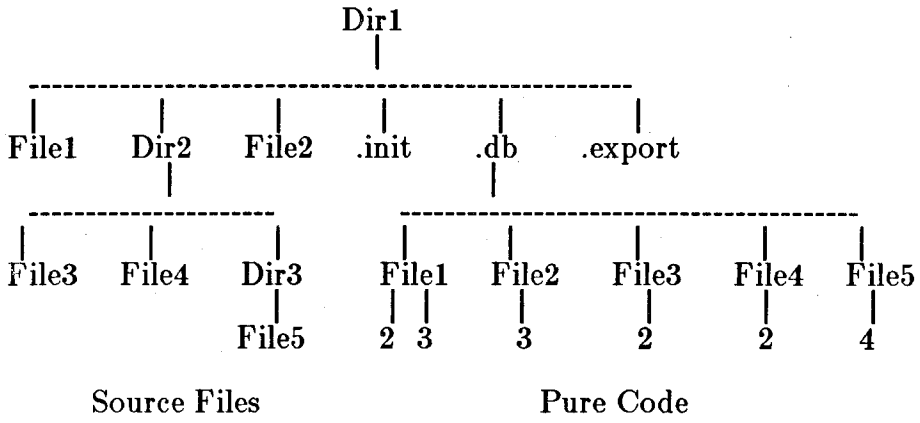

Figure 3.1 WUP 1.4 Module - File Structure

represented by an abstraction which ignores the details that are not necessary at this level. The abstraction is expanded at some lower level. A hierarchical organization clearly shows the precedence and dependence relations between the program components becoming itself an integral part of the program documentation.

A module, in general terms, should constitute the smallest separately compilable unit of a program. Therefore, in WUP 2.0 each source file has been made into a separate module as opposed to only directories as modules in the old system. A source file may now contain any number of different predicates, with the restriction that all clauses for the same predicate must be grouped together. For example, if a file contained the predicates p and q the clauses for p and q could not be intermixed. The file **p <- q; q; p;** is invalid, since q divides the two clauses of the predicate p, and will produce an error message indicating a duplicate predicate name, p. It should be written as **p <- q; p; q;.** Predicates with the same name but different arity, can be considered to be different predicates and need not appear together. The file **p(X); q; p(X Y);** is valid. This restriction is necessary for the compilation phase, as will be explained later. Associated with each file are the special files **.db, .dict, .export, .lock**, and **.init** each prefixed with the file name. The special files for the source file **test** would, for example, be named **test.db**, **test.dict,** etc.

The value of modules is, as mentioned, that they can be combined to form larger program units. This is accomplished by making each UNIX directory a super-module to all modules (i. e. files and sub-directories) contained within it. A super-module is a means of grouping a number of related modules into a single unit which can then be treated as one module. In this way a hierarchical module structure is created statically, by the programmer, through the UNIX file structure. Figure 3.2 shows an example of the organization of a number of Unix directories and files. Each file in the tree is a separate module and each directory a super-module combining all modules below it into a single module. The directory **Dir1** is a super-module containing the modules **File1, File2,** which are source files, and the directory **Dir2. Dir2** is a super-module consisting of the modules **File3, File4,** and **Dir3. Dir3** is a super-module consisting of the module **File5.** It is important to note that, except for certain built-in predicates, WUP 2.0 treats modules and super-modules in the same manner. This entire structure can now be referred to as a single module under the name **Dir1.** At the same time each element in the tree can be referred to individually as a separate module. The user organizes his program modules simply by organizing the files under UNIX directories.

Associated with each directory are the special files **.export, .lock, .order,** and **.init.** [1] These files must reside in the first level under the directory and are not prefixed by any name. In the example, the file **File1.init** applies to the module **File1** while the files **.init** and **.export** , at the same level, apply to the super-module **Dir1.**

3.2. Compiling

All WUP modules, including super-modules, can be separately compiled into pure code. When a super module is compiled all modules subordinate to it will be compiled. Modules which are source files are only recompiled if they have been altered since the last compilation. When a source file module is compiled the corresponding files **.db,**

[1] The files .db and .dict are needed only for the compiling of source code and therefore do not apply to a directory.

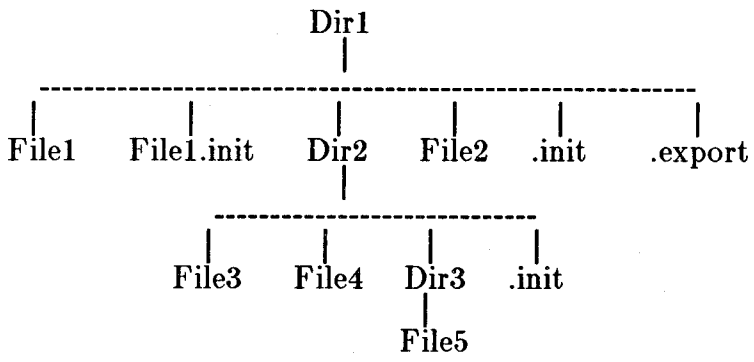

Figure 3.2 WUP 2.0 Modules - File Structure

containing the pure code, and **.dict,** containing information concerning **.db,** will be created. The file **.dict** will contain a clause, for each predicate in the source file, of the form **dict(name,narg,ncl,offset)** where **name** is the predicate name, **narg** the number of arguments or arity, **ncl** the number of clauses, and **offset** the offset in bytes of the corresponding pure code from the start of the **.db** file. This provides the information necessary for retrieval of the pure code.

As each predicate is read from the source file the pure code is generated and written to the **.db** file and a dictionary clause is created. If a syntax error is detected in the source code, pure code generation will cease, however the remainder of the source file will be checked for further syntax errors. When all errors have been reported, the user will be asked to edit the source file. Since any part of the file may be altered during editing, the entire file will be recompiled. If the file is not edited, WUP will terminate and no pure code file will be created. After compilation, one copy of the dictionary will be written into the **.dict** file and one copy will be retained internally to be used in accessing the **.db** file. If the file is not being compiled (i.e., either when the **-n** option is specified or the module is imported) then the **.dict** file will be read into memory.

To retrieve the clauses of a predicate, the offset from the dict entry is used to skip to the location of the first clause in the pure code file, and the number of clauses specified by ncl are read from the file. Without the restriction that all clauses for a single

predicate appear together in a source file the pure code for these clauses could be scattered throughout the pure code file, making the retrieval process unnecessarily complicated.

Each module in WUP is given an internal alias which is used in referencing it. The root module, used on the WUP command line, will be given the alias **usr.** If the root is a super-module, then for all modules contained in it, the file or directory name will be used as the alias. The built-in predicate **library** will display the module structure of the program, showing the module alias followed by the UNIX pathname of the file or directory, with modules indented to indicate the hierarchical relationships. The alias of each super-module will be followed by a '/' to high-light the fact that it is a directory and not a source file. Figure 3.3 shows the result of executing **library** when **Dir1** from Figure 3.2 is the usr module. Each module should be given a unique alias, otherwise a warning, indicating duplicate module aliases, will be issued.

```
usr/  (/u/cdbaird/Dir1)
      File1 (/u/cdbaird/Dir1/File1)
      Dir2/(/u/cdbaird/Dir1/Dir2)
            File3 (/u/cdbaird/Dir1/Dir2/File3)
            File4 (/u/cdbaird/Dir1/Dir2/File4)
            Dir3/(/u/cdbaird/Dir1/Dir2/Dir3)
                  File5 (/u/cdbaird/Dir1/Dir2/Dir3/File5)
      File2 (/u/cdbaird/Dir1/File2)
      imp1 (/u/cdbaird/File6)
imp2 (/u/cdbaird/File7)
man  (/u/prolog/manlib/man)
std  (/u/prolog/stdlib/std)
sys  (/u/prolog/syslib/sys)
```

Figure 3.3 WUP Library

3.3. Dynamic Structuring of Modules

The previous version of WUP allowed modules to be added to a program library dynamically by means of the built-in predicate **import.** When the predicate **import(Alias Pathname)** was executed the module corresponding to the given Unix pathname was inserted into the list of library modules just before the system modules.

This predicate has been retained and a new version added that permits modules to be inserted anywhere in the module hierarchy by specifying the parent module of the imported module. When the predicate **import(Parent Alias Pathname)** is executed the module corresponding to the given Unix pathname will be inserted in the tree structure at the end of the list of modules subordinate to the given parent module. In order to maintain the distinction between super and subordinate modules, only super-modules may be specified as a parent module. Specifying a source file module as parent will result in an error message. In Figure 3.3 two modules have been imported, imp1 using the predicate import(usr imp1 /u/cdbaird/File6) and imp2 using the predicate import(imp2 /u/cdbaird/File7).

Any module, including super-modules, may be imported in this way. The import predicate gives the user complete control over the structuring of his program modules.

3.4. Searching for a Predicate

When a predicate is called a search must be made for the corresponding pure code (see Appendix A). Searching may be done in one of two modes: via a normal call, which causes the entire library to be searched, or via the prove predicate, which restricts the search to a specific module. For each module, corresponding to a source file, WUP maintains an internal database, an external database (i.e. the **.db** file) and an auxiliary database into which clauses may be asserted during execution of a program. Although a super-module has no internal or external database it may have an auxiliary database. When a predicate is called for the first time, its clauses are loaded from

the external database into the internal database. Searching a module for a predicate involves first checking the auxiliary and internal databases then, if an entry for that predicate exists in the module's dictionary, retrieving it from the external database.

A general search for a predicate will systematically search the entire program library for the first instance of the predicate. The library is searched in an ascending breadth first manner starting with the module in which the call is made (see Appendix A). All sibling modules are organized into circular lists and the elements of the list are searched in the order shown by the library command. The usr module, all modules imported without a parent, and the system modules **man, std,** and **sys** exist as siblings at the highest level of the hierarchy. When a super-module is searched, since it combines all modules below it in the hierarchy into a single module, all its subordinate modules will also be searched in a breadth first manner. As an example consider the library shown in Figure 3.3. If a call is made in module File4, the library will be searched in the following order: File4, Dir3, File5, File3, Dir2, File2, imp1, File1, usr, imp2, man, std, sys.

A specific search occurs when the predicate **prove(Module Predicate)** is executed with the first argument bound to some existing module. Only the given module will be searched for the predicate and no other part of the library. When the given module is a super-module, again, all modules subordinate to it will also be searched breadth first. In the example, if module Dir2 were specified, the following search occurs: Dir2, File3, File4, Dir3, File5. If module File5 were specified then only File5 would be searched.

3.5. Special Files

During compilation WUP will ignore any file containing a period in its name. Certain files, as noted previously, are recognized by WUP as special files. The files **.db** and **.dict** are maintained by WUP, while all others are maintained by the user. When one of these files is associated with a super-module it will apply to all its subordinate

modules. This follows from the concept that a super-module combines all its subordinate modules into one module.

As an example, consider a file **.lock** at the first level of Dir2. This will have the effect of locking the modules Dir2, File3, File4, Dir3, and File5. Similarly a **.export** file under Dir2 will specify the predicates that are exported from all modules subordinate to module Dir2. Suppose module File3 contains predicates P and Q, and that P is exported by Dir2. A call to Q from module File1 will fail, however a call to Q from module File4 will succeed since Q is not hidden from this module. In order to hide predicate Q from module File4, an export list must exist for module File3 which does not include Q. Note that when a call is made from a subordinate module the auxiliary database of its super-module will be searched for the predicate without reference to the export list of the super-module. Also, if a predicate is exported by a super-module then, unless it is in the auxiliary database of the super-module,it must also be exported either implicitly or explicitly by some subordinate module in order to be visible outside the super-module. A predicate is implicitly exported when no export list exists and all predicates in the module are exported.

The file **.order** is used to specify the ordering of files when creating a listing of a program using the built-in predicate **make_list** and is relevant only for super-modules. It contains clauses of the form **order(alias)** where **alias** is the module alias of the file. If the alias refers to a super-module, then the corresponding directory should contain its own **.order** file specifying the modules contained in it that are to be listed. For example, to make a program listing of all files under Dir2 (Figure 3.2) the **.order** file should contain the following clauses :

```
order(File3);
order(File4);
order(Dir3);
```

Dir3 should contain a **.order** file with the clause order(File5). **Make_list** will create a

listing file containing the three source files, File3, File4, and File5, in that order, each preceded by its alias, full UNIX pathname, and a list of its exported predicates. In addition, the alias, UNIX pathname, and export list of each super-module, in this case Dir2 and Dir3, will precede and follow the listing of its component source files.

3.6. Built-in Predicates

As a result of the changes in the structure of WUP programs a number of changes were necessary to some of the built-in predicates and the behaviour of others required clarification.

All the predicates that deal with editing and compiling single predicate source files are now no longer valid. In keeping with the concepts of modular programming, the entire module should be regarded as the unit that is being altered rather than an individual predicate, therefore only modules may be specified in edit and compile commands. The predicate **rc,** used for recompiling such source files, has been deleted. The predicate **compile** can be used for compiling any module. The predicates **vi, ed, vic, edc** now take only one argument, a module which corresponds to a source file. **Compile** will check the time stamp on the file against its pure code file and only compile those files that have been altered since the last compilation. When a module is recompiled the main database will be cleared. **Vic** and **edc** call **compile** to recompile the file after it has been edited.

A **.order** file, specifying the files to be listed with the **make_list** command, can only be associated with a directory, or super-module. Consequently the predicates **make_order, consult_order** and **retract_order** take only super-modules as arguments. When **make_list** is given a super-module as argument a .order file must exist under the specified directory. **Make_list** can also be used to produce a listing of any individual source file.

The predicates for listing the clauses in a module, **list_main** and **list_aux,** when applied to a super-module will not list the predicates in the subordinate modules. This is an exception to the rule that all modules are treated equally. **List_aux** will list the clauses in the auxiliary database of the super-module while **list_main** will have no effect since a super-module has no main database. The predicates **clear_main, clear_aux** and **show** will operate in the same fashion.

Assert and **retract** or **delax** can be used for adding or removing clauses from the auxiliary database of any module. **Retract** and **delax** will not search the subordinate modules of a super-module for a clause. **Retrieve,** which searches a module for a given predicate, will however, when applied to a super-module, search the subordinate modules as well.

3.7. Comparison

A test program was run under both the new and old versions of WUP to provide a comparison of the two in terms of storage space and execution speed. The program chosen for the test was Debugger, a relatively large system designed to discover bugs in Prolog programs. In both versions it consists of several modules.

The overall saving of storage, including the system libraries, was dramatic as Figure 3.4 illustrates. The figures shown include both the source code and compiled code. Approximately two thirds less disk space was used under the new version of WUP. The debugger program itself was only slightly more than one fifth the size of the old version.

Informal testing of the execution of the two versions showed no noticeable difference in the performance. The execution speed in LIPS (logical inferences per second), found using the WUP built-in predicate **time,** was roughly equal.

	WUP 1.4	WUP 2.0
debugger	390	85
manlib	291	289
stdlib	298	99
syslib	356	33
total	1335	506

Figure 3.4 Disk Usage (kbytes)

4. Using Modules in WUP

DeRemer and Kron [4] make a distinction between programming in the large and programming in the small. Programming in the small refers to programs that are simple enough to be completed by one person and contained in a single module. Programming in the large involves the structuring of large collections of modules. Many of the programs written by WUP users fall into the former category. When programming in the small, users can simply ignore the modular features of WUP. Programs can be contained in single files, which the user need not relate in any way to the concept of a module. This will make programming easier for naive users than in the previous version of WUP since they need no longer concern themselves with modules or deal with programs spread over numerous files and directories. It resembles more closely the format of the programs they may be used to writing in other languages and hence may be less confusing.

When programming in the large, users should keep the following guidelines in mind. A module should contain only related predicates which implement a single program task corresponding to a procedural or data abstraction and should be limited to perhaps a dozen predicates and one or two pages of code, for efficiency as well as to reduce the complexity of the program. Any module larger than this can probably be divided into a number of smaller tasks.

The structuring of modules into programs is achieved through the organization of source files among Unix directories. The corresponding WUP library will be automatically constructed. As programs become larger and more complex, using modules from several sources or written by several programmers, the program library can be dynamically configured using the import predicate. There is a natural progression, reflecting increasing program complexity, from single module programs, to statically structured libraries of modules, and finally dynamically structured libraries.

Every module, including super-modules, should have an export list containing as few predicates as possible. If a module implements a procedural abstraction, then only a single predicate representing the procedure call should be exported. If a module implements a data abstraction, then only those predicates representing the operations on the data type should be exported. In both cases any other subordinate predicates used in the definitions remain hidden. The default of exporting all the predicates in a module should not be used except in the case where a module is a collection or library of commonly used routines such as the WUP sys and std libraries . By explicitly specifying each module interface in this way, the user gains the advantages of information hiding.

Using export lists and limiting the number of predicates in a module can increase the efficiency of WUP programs by reducing the search time for finding predicates. Most calls should be made to predicates within the same module as the call. Each such call causes a linear search to be made of a list of the predicates in the module. When a call is made to a predicate outside the module, the export lists of other modules are searched until the predicate is found. If a module has no export list, then the entire list of predicates in the module must be searched, thereby increasing the search time which in large programs could be significant.

Users should keep in mind that modules are meant to be independent and reusable program units. Each module should be separately implemented and tested. Only when individual modules have been certified should they be combined. Testing a complete program should involve only the testing of the module interfaces, not the modules themselves.

Programs written for the old version of WUP may be run on the new version without alteration. Each predicate file will be taken as a separate module, however the predicates will still be united in one module under the directory as before. The special files associated with the directory, .export, .order, .lock, and .init will still be valid and will produce the same results. However, to gain the space efficiency of WUP 2.0, it is advisable to reconfigure WUP 1.4 programs.

5. Modules in other Logic Programming Systems

5.1. M-PROLOG

M-PROLOG is a PROLOG system that is based on the concept of modules. An M-PROLOG program consists of one or more modules with one module designated as the root or main module. A module consists of a name, an interface specification, a set of declarations, a set of predicate definitions, and a single goal. The main module is activated when the program begins and its goal becomes the program goal. Other modules may be activated and deactivated using built-in predicates.

The module interface specifies the interaction of the module with other program modules. Specifications can be made regarding the visibility of all names used in a module, where a name is any constant including predicate names. A name declared as **local** can be used only within the module and, conversely, a name declared as **visible** can be used both inside and outside the module. Similarly a name declared as **hidden** is not visible outside the module while a **global** name is. All names are converted to a internal code by the system. This can be controlled by declaring a name as **symbolic**

which means that it will not be coded, or as **coded**. Declaring a name as **global** will also leave it uncoded, while names declared as **hidden** will be coded. The declarations local and visible have no effect on the coding of names. Suppressing the coding of a name allows the user to preserve its original form for printing or some similar purpose. The general specifications **all-visible, all-symbolic,** and **all-global** apply to all names in a module. These will be overridden by any other specifications.

In addition to controlling the visibility of names, one can specify **import** and **export** lists, that apply only to predicates. Both the predicate name and its arity are needed in the specification. Exporting a predicate makes the predicate name visible outside the module but not the names used within the predicate definition. To use a predicate, outside of the module in which it is defined, it must be exported by that module and imported by the module that uses it. Predicates should be exported rather than declared as visible, since this forces the system to make checks for its existence that are otherwise not done. A predicate that is not defined in a module cannot be exported by it and one that is already defined in a module cannot be imported by it.

5.2. micro-PROLOG

Micro-PROLOG is a Prolog system, designed for micro-computers, which also provides support for modular programming. The basic structure of a micro-Prolog program is a dictionary which lists all constants or names in use in the system. Each constant is represented as a pointer to a dictionary entry. The module structure is built on this by dividing the dictionary into separate segments. A module consists of a name, a set of clauses, import and export lists, and a dictionary. Each module is assigned its own segment of the dictionary when it is loaded. All constants in a module are, by default, local to it. When the same constant is used in different modules it will be entered in different segments of the dictionary and each entry will be, in effect, a different entity.

Names are made visible outside of a module by adding them to a module's import or export list. A name is made visible to a module when it is imported by that module and exported by the module in which it is defined. A module cannot export a name that is already exported by some other module. All file names, module names, and predicate names must be distinct.

When micro-PROLOG is loaded a system defined root module is made the current module. The root imports all names exported by all other program modules allowing all modules to be accessed from the root. The current module can be changed, however, other modules can only be loaded when the root is the current module.

5.3. HIMIKO

Himiko [6] is a logic programming system that goes the furthest towards realizing the concepts of modular programming, as stated by Ghezzi and Jazayeri, in a Prolog based system. It includes constructs for implementing data abstractions in modules as well as procedural abstractions. There is a strong correlation between the features provided by Himiko and those in imperative languages designed with abstraction capabilities.

Himiko allows the user to define data types, where a data type is a collection of terms. Two forms of data type are provided, **types** and **patterns.** **Types** are terms whose structures are hidden inside a module and that can be accessed only through a restricted set of operations. **Patterns** are terms whose structure is visible outside the module. **Types** correspond to the data objects defined by an abstract data type and are represented by functors. A functor considered as a data structure is analogous to a record in other languages, where the functor name equals the record name and each term in the functor equals a record field. Only the functor name is visible outside the module, the actual component terms may not be referenced by any other module. For example a **type** representing a state in the simulation of some process could be defined by the functor

state(waiting(<queue>), blocked(<queue>), active(<process>))

where **queue** and **process** are also **types** defined in separate modules.

Himiko types are equivalent to the concept of the private type in ADA and the opaque type in Modula, while patterns are non-private types. The details of a private type are not accessible outside the Ada package or module, in which it is declared. Variables in other modules may then be declared to be of a private type. Since the structure of the type is known only within the module in which it is defined, only the operations defined by the procedures in that module may be used to manipulate objects of that type. This is also the intent of the Himiko type.

An Himiko module consists of an interface and a realization corresponding to the ADA specification and body of a package and the definition and implementation of a module in Modula. The interface specifies the module's abstract data types and the relations that are visible outside the module. When a module implements an abstract data type, these relations correspond to the operations that are defined on the data type. Within the interface, the arguments of the predicates are written only as types. A predicate that creates a state in the previous simulation example would be written as **create_state(<state>).** The realization consists of a representation section and a clause section. In the representation, the structures of the abstract data types are defined. The clause section contains the logic program. Figure 5.1, taken from [6], shows an example of a module which implements a state as an abstract data type. An Himiko program consists of a hierarchy of modules.

5.4. Comparisons with WUP

The primary shortcomings of WUP are that it does not allow complete independence of modules and complete specification of module interfaces nor does it provide constructs for data abstraction. These characteristics adversely effect its value as a modular programming system.

```
module state_process_module
 interface
  type <state>

  rel  create_state(<state>);
       get_active_process(<process_id>, <state>);
       new_active_process(<process_id>, <state>, <state>);
        ...
        ...

 realization
  repr
       state(waiting(<queue>),
             blocked(<queue>),
             active(<process>))

  clause
       create_state(state(waiting(Q1),blocked(Q1),
                    active(self))
            <- create_1(Q1)
               create_1(Q2);

       get_active_process(ID,state(_,_,active(ID)));

       new_active_process(ID,state(W,B,_),
                          state(W,B,active(ID)));
        ...
        ...

end-of-module
```

Figure 5.1 State_process module

WUP's module interface is only partly specified by the export list. Access to any predicate should be by mutual agreement between its creator and its user. MPROLOG and micro-PROLOG ensure this by requiring both import and export lists. No predicate, defined outside of a module, can be accessed unless it has been imported by that module and exported by some other module. Requiring import as well as export specifications provides better control over module interaction, an improved internal program documentation, and the possibility for more extensive checking to be carried out, as is done in MPROLOG. MPROLOG and micro-PROLOG also allow the user to control the visibility of names other than predicate names. However, the actual value of this and the consequences of not having this feature in WUP are not clear.

In both Ada and Modula, the specification of imported elements also indicates from which module the element comes. While MPOLOG does require this, micro_prolog does not and, consequently, has the restriction that no predicate may be exported by more than one module. This eliminates the possibility of redefining predicates except locally in a single module. The redefinition or overloading of operations is a useful facility in the creation of abstract data types. Conceptually similar operations, addition of complex numbers and addition of integers for example, can be made syntactically similar, i.e., use the same predicate name.

WUP allows the redefinition of predicates but in an unrestrained manner. The scope of an exported predicate is determined only by its position in the program hierarchy. This creates the potential for unintentionally redefining a predicate causing problems similar to the problem in block structured languages of creating a hole in the scope of a global variable by re-using the variable name in some inner block. The explicitness of the MPROLOG interfaces eliminates the possibility of such errors and also improves the internal program documentation.

Only Himiko provides constructs for true data abstraction and information hiding. It allows data types to be defined within a module and their underlying structure to be concealed from other modules. The other systems allow the implementation of procedural abstraction but are inadequate for data abstraction. To provide further information hiding, the interface and realization components are made distinct, separating what is hidden from what is visible. In Modula and Ada, these parts can be compiled separately permitting changes to be made to the implementation without the possibility of corrupting the interface to the rest of the program.

WUP allows the organization of modules into larger program units to be controlled by the user. MPROLOG and micro-PROLOG have no provision for this. This gives the WUP programmer direct control over the structure of his program and to some extent control over its execution. The ordering of modules is similar to the order-

ing of clauses within a program and lets the user provide additional control information for the system.

Micro-PROLOG allows modules written in assembly language to be included in a program. MPROLOG is designed to permit modules to be written in other languages but this feature is not yet implemented. This facility is not incorporated in WUP at all.

6. Conclusion

Improvements have been made to the module system of WUP but it still does not meet all the criteria of a modular programming system. Abstraction, the basis of the module, is not fully supported. WUP has taken the simpler view that modules merely divide the code into smaller segments.

The problems of excessive storage requirements and the proliferation of files in WUP 1.4 have been eliminated. Any number of different predicates may now be written in the same file and each file constitutes a program module. The compiled or pure code now consumes only a fraction of the disk space it did previously. Users are now able to structure program modules in a meaningful way through a very simple mechanism. A hierarchical structure is automatically created directly from the Unix file structure. Modules can be inserted into this structure dynamically using an extended import predicate which specifies the parent module of the imported module.

WUP fails to provide adequate mechanisms for data abstraction, information hiding and, as a result, module independence. Module independence preserves the abstraction and guarantees the function of the module. By default, there should be no interaction between modules in a program, unless it has been explicitly defined. This ensures the complete independence of modules and requires the user to define, in detail, all interactions that can occur. Poor interfaces lead to the production of incompatible components.

WUP could be extended by allowing the use of modules written in other programming languages to be included in a WUP program. Many of the built-in predicates are written as C functions so perhaps a generalization of this mechanism could be used. The existence of these predicates is evidence of the usefulness, if not the necessity, of such modules.

References

[1] Barnes J.G.P., *Programming in Ada*, Addison-Wesley Publishing Company 1984

[2] Cheng M.H.M., *Design and Implementation of the Waterloo Unix Prolog Environment*, Research Report CS-84-47, Dept. of Computer Science, University of Waterloo 1984

[3] Clark K.L., Ennals J.R., McCabe F.G., *A micro-PROLOG Primer*, Logic Programming Associates Ltd., London, England 1982

[4] Clark K.L., McCabe F.G., *micro-PROLOG 3.0 Programmer's Reference Manual*, Logic Programming Associates Ltd., London, England 1983

[5] DeRemer F., Kron H.,*Programming-in-the-Large vs. Programming-in-the-Small*, IEEE Transactions on Software Engineering SE-2 (June 76) pp. 80-86

[6] Furukawa K., Nakajima R., Yonezawa A., *Modularization and Abstraction in Logic Programming*, New Generation Computing 1(2) pp. 169-178, Institute for New Generation Computer Technology 1983

[7] Ghezzi C., Jazayeri M., *Programming Language Concepts*, John Wiley and Sons, Inc. 1982

[8] Kowalski R., *Logic for Problem Solving*, North-Holland 1979

[9] Liskov B., Zilles S., *Programming with Abstract Data Types*, SIGPLAN Notices 9(4) (April 1974) pp. 50-59

[10] Liskov B., Snyder A., Atkinson R., Scaffert C., *Abstraction Mechanisms in CLU*, Communications of the ACM 20(8) (August 1977) pp. 564-576

[11] MPROLOG Language Reference Manual Release 1.5, Logicware Inc. 1984

[12] van Emden M.H., Goebel R., *Waterloo Unix Prolog User's Manual Version 1.2*, Logic Programming and Artificial Intelligence Group, Dept. of Computer Science, University of Waterloo 1984

[13] Wiener R., Sincovec R., *Software Engineering with Modula and Ada*, John Wiley and Sons, Inc. 1894

[14] Wirth N., *Programming in Modula-2*, Springer Verlag 1983

[15] Wulf W.A., Shaw M., *Global Variables Considered Harmful*, SIGPLAN Notices 8(2) (Feb. 1973) pp. 80-86

Appendix A - Search Algorithm

```
PROCEDURE search_proc(module, predicate, VAR pure_code)

 BEGIN
  search_parent(module, predicate, pure_code)
  IF NOT search_specific THEN
   WHILE (module <> NULL) AND (pure_code = NULL) DO
    BEGIN
     module := module.next
     search_children(module, predicate, pure_code)
     IF pure_code = NULL THEN
      BEGIN
       module := module.parent
       search(module, predicate, pure_code)
      END
    END
 END {search_proc}


PROCEDURE search_parent(module, predicate, VAR pure_code)

 BEGIN
  search(module, predicate, pure_code)
  IF pure_code = NULL THEN
    search_children(module.child, predicate, pure_code)
 END {search_parent}


PROCEDURE search_children(module, predicate, VAR pure_code)

 BEGIN
  start := module
  WHILE (pure_code = NULL) AND NOT (finish) DO
   BEGIN
    IF module.child <> NULL THEN
     search_parent(module, predicate, pure_code)
    ELSE
     search(module, predicate, pure_code)
    IF pure_code = NULL THEN
     module := module.next
    IF (module = start) then finish := TRUE
   END
 END {search_children}
```