AI-91585

Adding generic modules to flat rule-based languages: A low cost approach

J. Agustí-Cullell, C. Sierra and D. Sannella*

Centre d'Estudis Avançats de Blanes, CSIC
17300 Blanes, Girona, Spain
e-mail : AGUSTI@CEAB.ES and SIERRA@CEAB.ES
* Department of Computer Science, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, U.K.

Area: Expert Systems

ABSTRACT

We present a modular system for Rule-Based Languages as an application of a functional theory of modularity. The theory can be applied to modularize any flat rule-based language because it is practically independent of the internal nature of modules. The resulting system supports the construction of structured large Knowledge-Bases using generic modules and provides facilities for information hiding (data abstraction). The module language requires minimal alteration of the underlying flat rule-based language; it is in fact a metalanguage which can be easily implemented on top of the existing rule-based language. Main hints for the building of a compiler that translates from the module language to the flat language are provided.

1 INTRODUCTION

Many existing rule-based languages have few structuring facilities for "programming in the large". This is one of the reasons why the development of large applications is difficult. To improve this situation, there exist different proposals, from just adding some syntactic modular facilities to the rule-based language, up to the use of new object-oriented languages (SHWE87).

Our proposal lies in between. It is a semantic approach in the sense that modules have a natural connection with the underlying theory of the language, and it provides static parameterized modules that are close to objects in object-oriented languages. We have adopted this solution because it allows to keep unmodified our rule-based language MILORD (GLSV87) and, at the same time facilitates structuring the applications developed with MILORD. Furthermore, by this means we can experiment object-oriented concepts with minimum effort.

The functional approach to modularity that we present is based on ML modular system (HMM86). The same approach has been applied by (SW87) to logic programming. Other authors have also investigated modularity in the setting of functional and logic programming. (OK85, M86).

Some approaches to modularity require significant extensions to the interpreter of the flat language (i.e. (FGMO87)). To avoid this and to ensure decoupling of the module system from the underlying KBS language two restrictions are imposed:

1) modules should be declared before any reference to them is made, and

2) modules may not be created by rules.

The second restriction forbids to dinamically construct and to manipulate modules. If we leave this restriction out the result woud be a new KBS language in the object-oriented paradigm. In this case a module will become the equivalent of an object. This

Methodologies for Intelligent Systems, 4
Zbigniew W. Ras, Editor

aspect will be faced in the future.

But now, our main goal is to define a module system which can be easily implemented on top of any existing Knowledge Based System. That is, here we show how to build compilers which translate a module language program into a flat equivalent program.

In the rest of this introduction a functional approach to modularity is presented defining what our modules are and how they interact. The second section explains details of the module language by means of an example. The example is shown in Annex A. Finally, in section 3, some hints of the compilation process are given.

1.1 Modularity by functional abstraction

Modular programming is a strategy to reduce the difficulty of designing, verifying and modifying a program. This can be made by structuring the program as a number of components called modules with precisely defined interconnections or interfaces. To make the interfaces explicit between modules, a standard technique called functional abstraction will be used. This standard technique consists of isolating a piece of program or module from its context and then abstracting it by specifying :

1) Those modules which the abstracted module may depend upon (requirements or import interface).

2) The contribution of the abstracted module to the rest of the program (results or export interface).

The internal definition of this abstracted module is made in terms of the import interfaces.

The obvious example of this technique is functional programming, where such abstractions form the basic program units. The function body defines how to compute the output (results) in terms of the input (requirements). For modular programming we abstract encapsulated sets of the underlying language primitive declarations. Such abstractions are in fact program-valued functions and are called parametric or generic modules (the parameter's type being the import interfaces). When applied to particular modules that satisfy their import interface they result in a new module which satisfies their export interface. The method for building large KB systems consists in applying generic modules to previously built particular modules .

The functional approach to modularity does not prescribe the internal nature of the modules themselves. So, we can adapt this general approach to any flat rule-based language. From here on, we will call **modules** to the encapsulated sets of primitive rule-based language declarations (mainly fact and rule declarations) and call **generic modules** the parameterized modules. The interfaces will be called **specifications**, suggesting that they are the first step of the programming process.

Summarizing, modules are the basic units and are hierarchically organized . A generic module is a parameterized unit with explicit specification of parameters. Specifications describe the information that a module provides to the external world. Specifications, modules and generic modules are declared with explicit names (Annex A contains an example of modular program).

1.2 Modules and their interactions

To apply the mentioned theory of modularity to any flat rule-based language it must be precisely decided what modules are and how they may legitimately interact. The criteria chosen here are oriented to: facilitate the program construction, avoid common programming errors, support information hiding and maintain the module language decoupled from the flat language, in order to keep it unmodified.

We consider that a structured rule-based system is built up from individual modules, and each one consists primarily of a set of rules that define facts or execute actions. Interactions between modules arise by means of references to facts in the if-

part of rules. To manage such interactions the set of visible facts in a module must be controlled. In this respect a module is considered well-formed only when every fact used in the if-part of rules is either : 1) defined inside the module, 2) user asserted and declared as being dynamic or 3) imported from other modules. In this way the module language contributes to detect errors faster than post-processing facilities and it also allows to define safe name spaces.

2 THE RULE-BASED MODULE LANGUAGE

In this section the major elements of the module language are introduced by means of an example. The example shows a very simplified structure of an expert system application developed for the diagnosis of pneumoniae (PNEUMON-IA) (LSSV87). It appears in the Annex A and it will be referred from here on by numbers between brackets (i.e. the module *Bact-Lab-<24h* in Annex A will be referred as [10]).

2.1 Syntactic aspects

2.1.1 Primitive declarations

The flat rule language is considered to have no restrictions in the visibility of facts. That is, any fact is visible from everywhere. In the module language we consider two kind of fact declarations: dynamic and exportable. Dynamic facts are asserted by users at run time; those available inside a module are declared by **Dynafacts** $fact_1, \ldots, fact_n$. Exported facts are facts allowed to be used by other modules; they are declared by **Export** $fact_1, \ldots, fact_n$. All exported facts must be conclusions of rules in the module. Conclusions not mentioned in **Expofacts** are hidden. The example in Annex A shows different modules with both kinds of declaration.

The rule declarations in module language are kept exactly the same as they were in flat language. So all rules of existing expert system applications written in flat language can be reused.

2.1.2. Structure declarations

We have no restrictions on flat rule language. All structures defined in the module language will be compiled into rules. Module language introduces three kinds of declarations : set, specification and module.

Set declaration

Module language allows to give a name to a set of facts. The example [1, 2] in Anex A shows two fact set declarations : *laboratory and exploration*. The conceptual structure of a domain can be expressed with set declarations and operations on sets. From the point of view of compilation they act as read-macros that facilitate the writing of **Dynafacts** and **Expofacts** declarations.

Specification declaration

Specifications define the language which modules provide to the external world. The example [3-8] in Annex A shows different specification declarations. Later in the example there are module declarations instantiating them. The module declaration *Bact-exp-coma* [12] is an instance of the specification *Bact-exp* [5].

Specifications reflect the same submodule structure that modules have. For example, the specification *Bact-Lab* [4] reflects the dependence of *Bact-Lab-<24h* [10] on module *Definitions-lab* [5] by means of the declaration: **Module** X: *Def-Lab*.

Specifications can be inferred from modules in a direct way. When a module is declared with an associated specification identifier (i.e. *Bact-Lab-<24h: Bact-Lab*) the specification inferred from the module (i.e. *Bact-lab-<24h*) must match the explicit specification (i.e. *Bact-Lab*). A specification S_1 matches S_2 if a) S1 has a subset of the dynafacts of S_2 and a superset of the expofacts of S_2, and b) exported submodules by S_1 are a superset of exported submodules by S_2. Were the inferred specification larger

than the explicit specification, the additional facts would be hidden (not visible from outside) in the resulting module.

The syntax of specifications is similar to that of modules as shown in Annex A.

Module declarations

They have the form:

Module *modidentifier* = *modexpression*

where *modexpresion* can be either

a) an encapsulated set of declarations with limited scope,

b) a module name previously defined, or

c) a generic module application.

a.- *Encapsulated declarations*:

The module *Bact-exp-coma* [12] is an example of an *encapsulated set of declarations*. It contains a **Dynafacts**, **Expofacts** and **Rules** definitions. The **Rules** component concludes about *Bacterianicity* over patients in coma.

b.- *Module names:*

Module names are used to reference previously defined modules. To represent the explicit dependence of module A on module B, we write:

module A = **begin** **module** X = B . . . **end**

This is the mechanism by which hierarchically structured K.B. are built. For example, *Bact-Lab-<24h* module [10] referres the previously defined module *Definition-Lab* by the submodule declaration **Module** X = *Definitions-Lab* and so it makes all names of *Definitions-Lab* accessible to *Bact-Lab-<24h* module. That is, facts defined in the submodule *Definitions-Lab* are used in *Bact-Lab-<24h* via prefix-qualified names such as *X -> Leukocytosis*. The prefix indicates the access path to the fact.

The prefixing serves to distinguish between different instances of the same facts which could have different definitions associated with them. For instance, the module *Bact-Lab-more-24h* [11] concludes *Bacterian* and *Atypical* based on data from cultures that were obtained more than 24 hours ago. Similarly *Bact-Lab-<24h* concludes the same facts based on data from cultures obtained less than 24 hours ago.

To make the facts of a module directly accessible within another module we declare it open by: **Open** *modidentifier*. For example in the module *Bact-exp-nocoma* [13] we declare **Open** *Bact-exp-nocoma*, then no prefix is requiered to reference the facts of the opened module.

On the other hand, the fact *left deviation* of the *Bact-Lab-<24h* module is declared dynamic by the **Dynafacts** declaration and is considered a fact to be asserted at run time. Dynamic fact declarations are very useful in pure hierarchies because they allow to explicitly define which conceptual subdomains (set of data) will be used, so the compiler can check them.

By default all conclusions of a module are visible from outside using prefixed names. The **Expofacts** declarations restrict visible conclusions to a given set. In *Bact-Lab-more-24h* module it would not have been necessary to declare **Expofacts** because all conclusions are exported. However, to always declare **Expofacts** is a good methodology, so that future modifications of the module will not introduce undesiderable side effects (i.e. adding a new conclusion that must not be seen from outside).

c.- *Generic module application.*

The application of a generic module to actual parameters is considered as a module declaration. This topic will be explained in the next paragraph.

2.2 Semantics Aspects

2.2.1 Generic Modules

A module language without generic modules has some limitations. To see it let us consider an example. In Annex A, *Definition-Lab* [9] module is an instance of *Def-Lab* [3] specification. *Bact-Lab-<24h* : *Bact-Lab* [10] refers to *Definition-Lab* [9] and so the former can be considered as an extension of it. Extensions of new instances of *Def-Lab* [3] specification would require the rewriting of new *Bact-Lab-<24h* modules. This rewriting seams unnecessary because *Bact-Lab-<24h* code does not depend on any actual *Def-Lab* specification instance. So *Bact-Lab-<24h* could be abstracted from any *Bact-Lab* instance, thus obtaining a generic module. Then, each extension of an instance of *Def-Lab* could be generated by applying the generic module to this instance, and no code rewriting process would be performed by the programmer. This use of generic modules is safe because the code of the generic module is written only once avoiding inconsitencies.

Bact [14] is an example of generic module definition. It can be considered as an abstraction of the following particular module :

Module *Pre-Bact* = **begin module** X = Bact-Lab-<24h **module** Y = Bact-exp-coma ... **end**

This very generic module *Bact* [14] takes as parameters any two modules which matches its parameter specifications *Bact-Lab* [4] and *Bact-exp* [5] and it returns a module matching the specification *Bacterianicity* [6]. Other examples of generic modules having *Bacterianicity* as their parameter specification are [15, 16].

To build particular K.B. modules we apply the generic modules to previously defined particular modules. For instance, the module resulting from the following nested generic module applications:

Pneumococ (Bact (Bact-Lab-<24h, Bact-exp-coma))

concludes *Pneumococcus* in the situation of less than 24 hours and coma. On the other hand, the same generic modules applied with different parameters would lead to a different module. For instance:

Pneumococ (Bact (Bact-Lab-more-24h, Bact-exp-coma))

allows to conclude *Pneumococcus* in the situation of more than 24 hours and coma.

The specification of parameters and results in a generic module is made in order to check the matching with the actual parameters and results. It is also a kind of documentation. For example, the specification of the module resulting from *Bact* [14] application must be *Bacterianicity* [6] which in turn is the specification of the *Pneum*[15] parameter. The result of applying *Pneum* is a module with *Pneumococcal* [7] specification.

2.2.2 Abstraction process

Abstraction is a technique used both to limit the interaction between modules and to obtain simple specifications containing a designer's controlled amount of information. We want to hide internal details of a module to other modules so that some alteration in that module will not require alterations in the other modules. In this way we ensure that the modules depend only on exported facts.

Facts used in a module that are not mentioned in the result specification are hidden. This can be used to do data abstraction. For example, in module *Bact-exp-coma* [12] the conclusion *dehydrated* of rules *r4*, *r5* and *r6* used as premise of rule *r3* is hidden. So, the module *Bact(Bact-lab-<24h;Back-exp-coma)* [14] can not use this fact as premise of its rules.

2.2.3 Incremental KB building

We want to support the process of incremental KB building. So whenever definitions in a module change, these changes must be reflected in the rest of the program. The way to do it is simply by repeating the module applications that refer to the changed module. This relinking process can be automatized by the compiler, so the user gets rid of this cumbersome task.

2.2.4 Sharing declarations

Interactions between KB modules occur via common submodules. For instance, given the modules:

Pneumococ (Bact (Bact-Lab-<24, Bact-exp-coma)) : Pneumococcal

Myco(Bact (Bact-Lab-<24, Bact-exp-coma)) : Mycoplasmal

a generic module with parameter specifications: *Pneumococcal and Mycoplasmal* can be defined as follows:

Module Diag-same-date(X: Pneumococcal; Y : Mycoplasmal) =

begin module U = X **module** V = Y

{Here a rule is assumed: it uses facts of submodule *Bact-Lab-<24h* common to parameters X, Y}

end

We build now our KB in the following way:

Module Pneumococcus-diag<24-coma =

Pneumococ (Bact (Bact-Lab-<24h, Bact-exp-coma))

Module Mycopla-diag-more-24-coma=

Myco(Bact (Bact-Lab-more-24, Bact-anam-coma))

Module Diag-same-Labdate = Diag-same-date(Pneumococcus-diag-more-24-coma,

Mycopla-diag-more-24-coma)

Suppose that the definition of *Diag-same-Labdate* needs the same instance of *Bact-Lab* for both parameters of *Diag-same-date*. For example, one reason can be that we want to use the same date for both parameters (less than 24 hours or more than 24 hours in our example). However, in the former example they are defined by generic module applications using different instances of *Bact-Lab*. To solve this problem we wish to impose a restriction on the parameters of *Diag-same-date*. Sharing declarations are used to do it. Sharing declarations are equalities between submodules and are declared after the parameters of generic modules (path equations). For example, our *Diag-same-date* generic module should be rewritten as follows:

Module *Diag-same-date*(X : Pneumococcal; Y : Mycoplasmal

sharing X-> X-> X = Y->X->X)

Where the path equation X -> X -> X = Y->X ->X indicates that the instances of *Bact-Lab* in the bacterianicity instance of X and Y must be the same. Finally the well formed K.B. would be:

Module Pneumonia-diag-<24-coma =

Pneumococ (Bact (Bact-Lab-<24h, Bact-exp-coma)

Module Micopla-diag-<24-coma=

Myco(Bact (Bact-Lab-<24h, Bact-exp-coma)

Module Diag-same-labdate =

Diag-same-date (Pneumonia-diag<24-coma, Micopla-diag-<24-coma)

3 COMPILATION PROCESS

To build the compiler it is necessary to structure the semantic relations between the module language and the flat language. To do so, semantic functions in the framework of denotational semantics are used. The identifiers (names) of the flat language are the semantic elements needed to define:
1) Semantic objects
2) Semantic operations
3) Semantic equations

3.1 Semantic objects

The proposed semantic objects are tables which translate names in the module language into names in the flat language. The process of compilation builds these tables by analyzing the program syntax. The number of the tables corresponds to the classes of identifiers allowed by the module language.

3.2 Semantic operations

To facilitate the writing of semantic equations some operations on the semantic objects are needed. For instance, the matching of two specifications is made by using fitting operations between tables.

3.3 Semantic equations

The semantic functions are defined equationally. They generate and modify the semantic objects from syntactic structures.

Notice that flat language determines the semantic objects. Once they have been stablished it is easy to adapt the semantic operations and the equations of our compiler to any language.

Generic modules are treated as macros; they keep their bodies as syntactic objects rather than as some sort of parameterized structure.

The result of the compilation process will both be a flat language code and some tables, which are built during the process. The whole compiler has been written in Vaxlisp on DEC VAX machines.

4 CONCLUSIONS

Here we have presented a module system for KBS based on a functional approach to modularity, and we have shown its applicability to any flat rule-based language. The system supports the construction of large KBS using generic modules and provides facilities for abstraction. The system includes a notion of well formed structured KB which avoids common KB programming errors. The module system requires no alteration of the underlying flat KB language because it is a metalanguage easily implementable on top of it.

Annex A: Example of PNEUMON-IA

```
[1] Set Lab = ( Leukocytes : integer;  Granulocytes% : [0 100] ; left_deviation :
              fuzzy)
[2] Set Exploration = (Pleuritic_pain : fuzzy ; Expectoration : boolean ;
              Instauration : [subacute, acute] ; Hypotension : fuzzy ; Cough :
              boolean ; State : [mild, serious, very_serious] ; Skin_fold : fuzzy;
              Tachypnea : fuzzy; Dry_tongue : fuzzy; Axillar_sweat: boolean;
              fever : fuzzy; Headache : boolean )
[3] Specification Def_lab =
      begin
        Dynafacts = Lab
        Expofacts = ( Leukocytosis : fuzzy ; Leukopenia : fuzzy;
                        Neutrophilia : fuzzy )
      end
[4] Specification Bact-lab =
      begin
        Module X : Def_lab
        Dynafacts = Lab
        Expofacts = (Bacterian : fuzzy ; Atipical : fuzzy)
      end
[5] Specification Bact-exp =
      begin
        Dynafacts = Exploration
        Expofacts = (Bacterian : fuzzy ; Atipical : fuzzy)
      end
[6] Specification  Bacterianicity = begin ... end
[7] Specification  Pneumococcal = begin ... end
[8] Specification  Mycoplasmal = begin ... end
[9] Module Definitions_lab: Def_lab = begin ... end
[10] Module Bact-Lab-<24h : Bact-lab =
      Begin
            Module X = Definitions_lab
            Rules
            r1 if X->Leukocytosis and Left-deviation then Bacterian is Possible
            r2 if X->Leukopenia and Left-deviation then Bacterian is Almost_sure
            r3 if X->Neutrophilia then Bacterian is Possible
            r4 if no X->Leukopenia and no X->Leukocytosis and no Left-deviation
               then Atipical is Quite_possible
      end
[11] Module Bact-lab-more-24 : Bact-lab =
      begin
        Module X = Definitions_lab
        Dynafacts = Lab
        Expofacts = (Bacterian : fuzzy ; Atipical : fuzzy)
        Rules ...
      end
[12] Module Bact_exp_coma : Bact-exp =
      Begin
            Rules
            r1 if  Instauration is acute then Bacterian is  Moderately_Possible
            r2 if  State is serious or very_serious and  Hypotension
               then Bacterian is  Quite_possible
            r3 if  Cough and no Expectoration and no Dehydrated
              then  Atipical is  Slightly_possible
            r4 if skin_fold then Dehydrated is  Quite_possible
            r5 if no Tachypnea and dry_tongue then Dehydrated is sure
            r6 if no Axillar_sweat and fever then Dehydrated is Quite_possible
      end
```

```
[13] Module Bact-exp-nocoma :  Bact-exp =
     begin
          Open Bact-exp-coma
          Rules
          r1 if pleuritic-pain then Bacterian is  moderately-possible
          r2 if no (State is serious or very_serious) and  Headache
             then Atipical is Possible
     end
[14] Module Bact( X :  Bact-lab ; Y : Bact-exp ) : Bacterianicity =
     begin
          Inherit  X
          Inherit  Y
          rules
          r1 if X->bacterian >= Y->bacterian
             then Bacterian = X->bacterian is sure
          r2 if X->atipical >= Y->atipical then Atipical = X->atipical is sure
          r3 if X->bacterian or Y->bacterian then Bacterian is  Almost_sure
          r4 if X->atipical or Y->atipical then Atipical is  Almost_sure
     end
[15] Module  Pneum( X : Bacterianicity ) : Pneumococcal =
     begin
     Inherit  X
     Expofacts = (Pneumococcus : fuzzy)
     rules ...
     end
[16] Module  Myco( X : Bacterianicity ) : Mycoplasmal =
     begin
     Inherit  X
     Expofacts = (Mycoplasm : fuzzy)
     rules ...
     end
```

References

(FGMO87) K. Futatsugi, J. Goguen, J. Messeguer, K. Okada
Parameterized Programming in OBJ2. *Proceedings of Ninth Int. Conference on Software Engineering. Eds. IEEE Comp. Soc. Press 1987 pp 51-60.*

(GLSV88) Ll. Godo, R. López de Mantaras, C. Sierra, A. Verdaguer
MILORD: The Architecture and the Management of Linguistically Expressed Uncertainty. To appear in *International Journal of Intelligent Systems.* 1988

(HMM86) R. Harper, D. McQueen, R. Milner
Standard ML.*Report ECS-LFCS-86-2* of Edinburgh University.

(LSSV87) R. López de Mantaras, F. Sanz, C. Sierra, A. Verdaguer
MILORD+PNEUMON-IA: Un outil. et une application en medicine. *Colloque Intelligence Artificiel et santé.* Toulousse 1987, pp 45-54

(M86) D. A. Miller
A Theory of Modules for Logic Programming. *Proceedings of 1986 IEEE Symp. on Logic Programming.*

(OK85) R. O'Keefe
Towards an Algebra for Constructing Logic Programs. *Proceedings of 1985 IEEE Symp. on Logic Programming. pp 152-160*

(SW87) D. Sannella, L. A. Wallen
A Calculus for the Construction of Modular Prolog Programs. *Proceedings of 87 IEEE Symp. on Logic Programming. pp 368-378*

(SHWE87) B. Shriver, P. Wegner (Eds.)
Research directions in Object-Oriented Programming.*MIT press* 1987