Partial Evaluation in Milord II: A Language for Knowledge Engineering

J.Puyol-Gruart, J.Agustí-Cullell, C.Sierra

CEAB
Camí de Santa Bàrbara
17300 Blanes, Girona, Spain.
Ph: 34 72 336101
Telex 56372 CEABL-E
Fax: 34 72 337806
e-mail: Puyol@ceab.es, Sierra@ceab.es, Agusti@ceab.es

SUMMARY: In this paper a new language for Knowledge Engineering is presented. Its main characteristics are: modularity, incremental programming of KB's, partial evaluation and modular uncertainty treatment. The partial evaluation is presented as a good mechanism to provide richer communication patterns of the system and the final users. It allows to give more explanatory answers that the classical inference mechanisms employed in expert systems. A rough description of the language MILORD II is presented stressing its modularity structures that allow to define generic modules and incremental programming operations between the modules. Some examples obtained from the applications developed using this language are presented. *KEYWORDS: Knowledge Engineering, Partial Evaluation*

1. Introduction

Looking at an Expert System (ES) as a *blackbox*, the standard behaviour we can see is the following: The system tries to reach a goal witch usually consists in deducing the certainty value of a fact asked by a user. If the system can deduce the value of this fact it gives the answer to the user, if not the answer is *unknown*.

In this paper we explain why this behaviour is rather poor and we propose a more informative input/output ES behaviour. Even if the system can not give a value to a fact, it has a lot of information related to the fact that can be useful to the user. In fact the standard behaviour of ES modelises only one aspect of the way human experts communicate. Two cases are usually not taken into account:

a) In the case the system is able to answer the user question, we could be interested in knowing other explored but not successful deductive paths to obtain a higher certainty value for the user question. It also could be useful to know conclusions which are deducible from the reached goal and communicate them to the user.

b) A more evident case of this poor communicative behaviour of ES is when they answer *unknown* to a question. Usually the system deduces *unknown* because the user has not given enough information to the system, i.e. he has not answered to all the questions the system has asked. Maybe the user has not answered these questions because he did not know the relevance of the question. It will be then much more informative if the system was able to deduce, not unknown, but the set of facts you should know in order to prove a value for the question.

That is why for the reasons explained above we will introduce an enriched communication scenario and the possibility of information to be reconsidered in order to avoid the unknown answer.

On the other hand, the expert who develops the ES can not look at the ES under development as a blackbox. In order to validate the system he needs to see an opened perspective of the running ES. The standard behaviour offered to the experts consists of a trace of the execution, that is, which is the rule the system tries to fire, which is the value of a deduced fact, and so on. This trace give to the expert an idea of the execution flow, but he needs to validate the KB for each case it proves. We propose a richer tool for the expert, consisting in a KB specialised for the current context under test.

1.1. Enriched Communication Scenario

As explained above, the behaviour of a classical ES is the following:

1st) The user asks a question to the ES for example P?

2nd) The ES uses its deductive and control knowledge to find a certainty degree for P.

3rd) The ES gives this certainty value back to the user.

Looking carefully at how experts communicate their knowledge and at their problem solving procedures we can find much more complex communication mechanisms. Sometimes experts can not reduce their interaction only to the communication of certainty values of predicates. For instance, when communicating, experts in medical diagnosis also need:

1) *To condition their answers*.

Suppose that it is not known if a patient is allergic to penicillin. A module deducing the possibility of giving penicillin can answer: *Penicillin is a good treatment from a clinical point of view if there is no allergy to it*. From a logical point of view, the answer could be:

{if no(allergy_penicillin) then Penicillin is very_possible}

2) *To give conclusions that have to be considered with the answer.*

If in a culture of sputum pneumococcus have been isolated then it is strongly suggested to make an antibiogram to the patient.

{Pneumococcus_isolation is sure, Make_Antibiogram is definite}

3) *To give conditioned conclusions to be considered with the answer.*

A treatment with ciprofloxacine is not recommended for breast-feeding women. However if a woman is on breast-feeding period then the treatment can be carried if the woman stops breast-feeding.

{Cipro is very_possible, if breast-feeding then stop_breast-feeding is definite}

4) *To give a more general answer.*

Imagine that Gram positive coccus are detected. An answer to the predicate pneumococcus is at that moment too precise and cannot be given, but at least the morphological classification can be answered, i. e. : {coccus is definite}

These types of communication can be modelled by the next schematic sets of formulas:

The possible sets of answers to a question *p*? can be:

1) $\{(c_1 \quad c_2 \quad \ldots \quad c_n) \varnothing p\}$

2) $\{p, c_1, c_2, \ldots, c_n\} \varnothing p\}$

3) $\{p, ((c_{11} \quad c_{12} \quad \ldots \quad c_{1n}) \varnothing r_1), \ldots, ((c_{m1} \quad c_{m2} \quad \ldots \quad c_{mn}) \varnothing r_m)\}$

4) $\{c_1, c_2, \ldots, c_n\}$

To model such communication protocols, we need to extend the ES answering procedure. What we need is to answer to a given question with a set of formulas (rules and facts). To answer the question, the rules considered are those in deductive paths to and from the question. The facts in the answer are those that have been obtained in the application of such rules. The rules in the answer are those which could not be applied because they used unknown knowledge. We only consider the rules in the deduction tree of the question because we assume that when an user makes a question it expects an answer, and eventually all the relevant information associated with it to that question only.

Example 1: Consider a module M with the following set of rules:

Rules = {**if** Gram_negative_rods **then** Treatment_with_ciprofloxacine **is** possible, **if** Treatment_with_ciprofloxacine **and** breast-feeding **then** Stop_breast-feeding **is** definite, **if** Bacterian_infection **and** Gram_negative_rods **then** Treatment_with_ceftriaxone **is** very_possible}

and the following set of facts:

Facts = {Gram_negative_rods **is** definite, breast-feeding **is** definite, Bacterian_infection **is** definite }

Given the question *Treatment_with_ciprofloxacine*? made by an user we can see the different answers obtained by a classical procedure and by the one we propose:

Classical answer = {Treatment_with_ciprofloxacine **is** possible}

Proposed answer = {Treatment_with_ciprofloxacine **is** possible, Stop_breast-feeding **is** definite}

The mechanism of computing classical answers is to look for a proof of the question. This proof will trivially be obtained by a modus ponens application:

if Gram_negative_rods **then** Treatment_with_ciprofloxacine **is** possible

Gram_negative_rods **is** definitive

-

Treatment_with_ciprofloxacine **is** possible

In our proposal, the set of rules related to Treatment_with_ciprofloxacine is obtained first:

{**if** Treatment_with_ciprofloxacine **and** breast-feeding **then** Stop_breast-feeding is definitive, **if** Gram_negative_rods **then** Treatment_with_ciprofloxacine **is** possible}

then, using the set of known facts, rules will be partially evaluated in order to obtain the reduced final set of formulas:

{Treatment_with_ciprofloxacine **is** possible, Stop_breast-feeding **is** definite}

Example 2: Consider the same set of rules of the previous example and the following set of facts:

Facts = {Gram_negative_rods **is** definite, breast-feeding **is** unknown, Bacterian_infection **is** definite}

then the answers will be:

Classical answer ={Treatment_with_ciprofloxacine **is** very_possible}

Proposed answer ={Treatment_with_ciprofloxacine **is** possible, **if** breast-feeding **then** Stop_breast-feeding **is** definite}

In this second case the classical evaluation provides the same result as in the previous example but in our proposal a different final set of formulas will be provided. This set is different because the fact *breast-feeding* is unknown and then the third rule cannot be applied:

{Treatment_with_ciprofloxacine is possible, **if** breast-feeding **then** Stop_breast-feeding **is** definite}

It is very important to consider that in a real case the answer can be very large and very difficult to understand. Then the most important point is to notice witch are the most relevant facts we should know in order to obtain a proof conditioned to the unknown information. A complexity criteria is used for the answers. When there is too much unknown information, the system gives to the user a summary witch allows him to reconsider the unknown information, and leads him to the solution.

The language MILORD II is an extension of a previous knowledge based language called MILORD (GOD 88), (SIE 89), used to develop several medical expert systems (VER 89) (BEL 91). We have to remark that MILORD II is a modular language, where each module is a partial knowledge base (KB). The communication scenario (presented above) is the same between the user and the ES that the communication among modules.

All these points will be developed along this paper. In section 2 a description of

MILORD II modular language is presented. The desired communication behaviour is obtained by an execution model based on partial evaluation. The overall description of this mechanism will be presented in section 3.

2. Milord II Modular Language

2.1. Introduction.

The experience in KB design, specially using knowledge acquisition techniques (PLA 89), allows us to detect a number of necessities that can be tackled with the methodology we propose here. Amongst them we can emphasise: Modularity, reusability, incrementality, local control, validation and multilanguage representation. Let us briefly discuss the meaning of this words.

Modularity. The usual way of understanding a complex problem is to decompose it into simple subproblems using simple operations. To make a useful decomposition of problems, subproblems must have a simple and well defined interaction.

To determine the adequate nature of modules, or partial KB's, that represent the subproblems and to define the operations of combination of these KB's is a key point in the design of a language for KE.

Reusability. In the building process of a KB it is important to be able to reuse existing partial KB's of problems solved beforehand (CHA 86) (GOG 86). For instance, although the diagnosis of infectious chest illness and that of chest tumours are essentially different, they could share the analysis of a thorax radiography. So, as a requirement of our language, we need generic program units that could be instantiated, or reused, in different contexts.

Incremental modification of KB's. The KB building methodology is an iterating two-step process. First a prototype is build (or modified), then it is validated. Thus, it is convenient to have some safe refinement operations that support this process of incremental KB building (modification). These operations have to preserve the adequacy of the KB behaviour with respect to the expert behaviour.

Local Control. Control is a component of the problem solving task tied to the domain knowledge. Thus it must be a component of each partial KB.

Validation. The problem of KB validation has been applied only to the KB considered as a whole, without taking into account its building process. It is then necessary to think about validation characteristics in the building process, i.e. in the different and successive partial KB's which, conveniently combined and progressively refined, will result in the total KB. The validation should not be just a final quality control test, but it must be integrated into the building process of the system.

Multilanguage representation. The basic operations of construction and modification are independent from the underlying language used to define the bodies of the modules. This independence allows the use of different languages of representation in the different modules (HAR 89). An easy example of this is the use of different multi-valued logics in each module (SIE 90).

On the modules two types of operations are defined: the composition or combination of modules into a new module which contains them as submodules, and the refinement of a module. The process of development of a KB is represented with a DAG in which the arcs symbolise both the operations of composition and the operations of refinement and where the nodes stand for the modules (the content of the nodes is explained in the paragraph devoted to the description of the language). The roots of this graph stand for the more general and less defined partial KB specifications, i.e. the body is not completely defined. The terminal nodes are the more concrete and executable KB's.

2.2. Proposed language: MILORD II

In this section the major elements of the language MILORD II are presented by means of an example. Most part of the example is artificial; it has been designed exclusively in order to introduce the syntax and semantics of the language. The example will be introduced progressively. The modules whose contents are not defined in the paper will be marked in italics.

2.2.1. Modules

The basic KB units written in MILORD II are the modules. These are hierarchically organised, and are composed of a set of importation, exportation, rule, control, meta-rule and submodule declarations. The Import/Export interface establishes the input/output behaviour of the module and the declaration of the submodules settles the hierarchic structure of the KB. The declaration of submodules is identical in every aspect to the declaration of the modules (see the example in figure 1).

The language provides three basic mechanisms of module manipulation:

1) Composition of modules through the declaration of submodules,

2) Refinement of modules, and

3) Composition of modules through operators defined by the user via generic modules definition.

The semantics of a module are, intuitively, a module identifier that can be referenced by the other modules and a piece of code whose main functionalities are shown below.

2.2.2. Primitive declarations

The most basic elements of MILORD II are the same as in MILORD (GOD 88), (SIE 89), the underlying current language of MILORD II. These are facts of order 0+, production rules and meta-rules. Figure 1 exemplifies the primitive declarations outlined in this section.

Import declarations

Imported facts are those whose values are obtained at run time from the user. These facts are declared by:

Import fact_1, fact_2,..., fact_n

and the values are obtained when needed in the evaluation of a rule. With this declaration we define the input interface of the module which contains it. The code of a module containing an import declaration will be allowed to ask for values of imported facts only. They will be obtained from the user.

Export declarations

Exported facts are those facts that can be used by other modules. All exported facts must be conclusions of rules in the module or else be imported by the module. They are declared by:

Export fact_1, fact_2, ..., fact_n

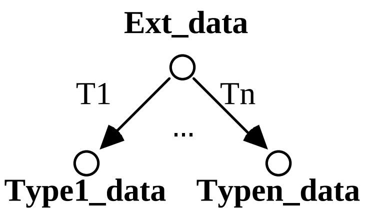
Conclusions of rules and imported facts not mentioned in the export declaration are hidden to the rest of the modules, i.e. they cannot be used in the body of the rest of the modules. This is the only mandatory declaration in the construction of a module. A module with no exported facts is meaningless. The code of a module containing an export declaration will provide means to answer questions about the values of the exported facts only. Also visible submodules will provide code with the same characteristics that will be added to the code of the module that contains them.

Kernel declarations

The kernel is made up of two components called *deductive knowledge* and *control knowledge*. Deductive knowledge includes the declarations of the object language which in our current implementation is a production rule language. Control knowledge is represented by means of a meta-language which acts by reflection over the deductive knowledge and the module hierarchy. The current implementation of the meta-language allows the definition of meta-rules and the definition of some control parameters (v.g. *evaluation type*). In the future the meta-language will allow a structured and incremental programming of the control. A module with an empty kernel can be considered to be a pure interface. The kernel of a module can be incrementally filled up by using the incremental programming primitive operation ":" (not developed in this paper).

The kernel provides the code of a module without any restrictions on the interface. In our case the code is basically a set of rules and meta-rules to be interpreted by an inference engine.

```
Module Ext_data =
    Begin
        Module T1 = type1_data
        ...
        Module Tn = typen_data
        Import fever, inmunodepressed
        Export ext_data1, ext_data2, ..., ext_datam
        Deductive knowledge
            Rules :
            R1 if T1/X1 and inmunodepressed
              then conclude ext_data1  is sure
            ...
            Rm if T1/X3 and T4/X5
              then conclude ext_datam  is quite_possible
        End deductive
        Control knowledge
            Evaluation is lazy
        Deductive control is nil
        Structural control is
            MR1 if no(fever) and no(T4/X3)
                    then eliminate(T1)
        End control
    End
```



T_1 to T_n are labels in the archs standing for internal names of pointed modules inside ext_data

Figure 1: Module definition and its graphical representation.

In figure 1 the text of a module declaration can be found. Bellow it the graph representation of the module is presented. A graphic, and intuitive, representation of combinations between modules will be used along the paper.

2.2.3. Module and submodule declarations

Modules (or submodules) have the form:

Module *modidentifier* [: $modexpr_1$][= $modexpr_2$]

where $modexpr_1$ and $modexpr_2$ can be either

a) An encapsulated set of primitive declarations and submodule declarations with a limited scope,

b) A module name defined elsewhere or even not yet defined, or

c) A generic module application.

$modexpr_1$ defines a module from which the module *modidentifier* is a refinement.

The symbol ":" stands for the module refinement operation and the symbol "=" for the module composition operation. If modexpr1 and modexpr2 are the empty string, the effect of the declaration is just to keep the modidentifier in the environment where the declaration is made. In the next sections we detail these forms of module declarations.

<u>Encapsulated declarations</u>

The module *Ex_data* (figure 1) is an example of an encapsulated set of declarations. It contains all the primitive declarations mentioned in the previous section and also the declaration of a set of submodules. Its semantics are those explained in the primitive declarations section.

<u>Declarations by reference: module composition.</u>

Module names are used to refer to other modules. The referred modules may not have been created in the moment when their names are used, expediting thus a top down design.

Module *Method_of_DA* (figure 2) references module *Ext_data* renaming it *B*. This declaration makes all facts exported by *Ext_data* be visible in the kernel of *Method_of_DA*. Thus, *Method_of_DA* rules can use these facts in their premises prefixing them with the identifier of the submodule which exports them. The symbol for prefixing is "/". The concatenation of prefixes allows to represent the path to a fact in the modular hierarchy. The prefix is useful to distinguish between different instances of the same facts in different submodules.

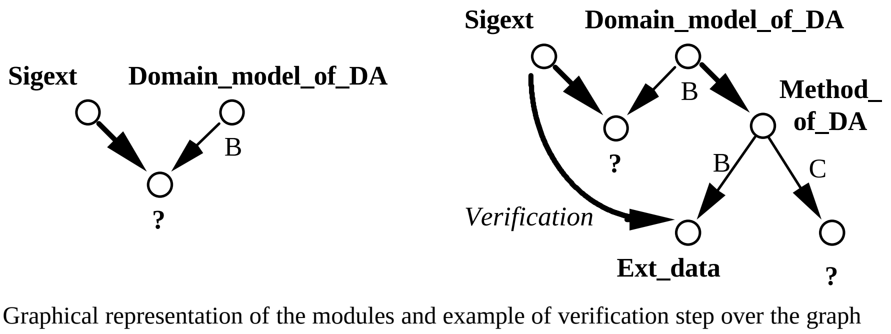
If we do not want to change the name of the module referenced in a submodule declaration, we can use the declaration **Inherit** (see module *DM_noexpert*). In order to make the facts of a module directly accessible without prefixing them we declare it **open**.

The formal semantics of hierarchical composition of modules is built up from two elementary operations: (1) The union of bodies of modules and (2) the renaming of name spaces that avoid the undesired interactions between module bodies.

The semantics of a submodule is a variant of the semantics of a module. The exported facts and submodules of a submodule will be visible outside the module declaration if the ':' operation allows it, i.e. if the submodule is not hidden by ':'. For instance in figure 2 the submodule *C*:*sigext* is not visible outside *method_of_DA* because *Domain_model_of_DA* does not contain any submodule called *C*. So, *Method_of_DA/C/P* can not be used. Despite that, inside the module *Method_of_DA* the submodule *C* will keep all its semantic functionalities, i.e. *C/P* will be allowed.

```
Module Domain_model_of_DA =
        Begin
                Module B : Sigext
                Export $DA_1$; ...; $DA_n$
        End

Module Method_of_DA :
        Domain_model_of_DA =
        Begin
                Module B = Ext_data
                Module C
                Deductive knowledge =
                        $R_1$ if B/$ext\_data_j$ and ...
                                then $DA_{i_1}$
                        ...
                        $R_m$ if B/$ext\_data_k$ and ...
                                then $DA_{i_m}$
                        ...
                End deductive
        End
```



Graphical representation of the modules and example of verification step over the graph structure of the modules.

Figure 2. Example of module refinement.

2.2.4. Declaration of generic modules

The instantiation of a generic module is considered as a module declaration. This topic will be developed in the section 4.5.

The definition of generic modules opens to the user the possibility of defining specific operations of composition. This standard technique consists of isolating a piece of program, or module, from its context and then abstract it by specifying:

1) Those modules upon which the abstracted module may depend (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 functional body defines how to compute the output (results) in terms of the input (requirements). In modular programming such abstractions are in fact program-valued functions and are called parametric or generic modules (the parameter type being the import interfaces). When applied to particular modules that satisfy their import interfaces, they result in a new module which satisfies their export interface. The method for building large KB systems consists of applying generic modules to previously built particular modules.

An example of definition of generic modules is showed in figure 3. That example is obtained from Bacter-IA, a medical application being developed in our laboratory using Colapses. There, the module *Global_gram* represents a general gram analysis over different samples, that have in common only those aspects established in the module *Sample*: the output interface. In concrete, modules such as Sputum,providing different views over the same exported facts can be defined. Keeping the common parts in a generic module we can save code and time and make the code much more understandable. Finally when a module is needed to make the gram analysis of an sputum sample, it is only necessary to put both modules together by a generic module application *Global_Gram(Sputum).*

To further determine the semantics of generic modules we can say that:

* The parameters of the definition are declared as refinements between names of formal variables and modular expressions. These modular expressions guarantee a minimal output interface to be used in the body of the generic module.
* When instantiating a generic module upon some concrete modules, the refinement operation declared in the parameters is carried out. Then, and with the help of code-expanding techniques, the resultant module is obtained.

* The instantiation of a generic module upon concrete modules can be restricted by a declaration of submodule sharing between the current parameters. That is, the submodules which have been declared as "shared" must be identical. (AGU 89).

We want to support the process of incremental KB building by means of generic modules. So whenever the definition of a module changes, these changes must be reflected in the rest of the program. The way to do it is just to repeat the module applications that refer to the modified module. This re-linking process can be automatised by the compiler, so that the user gets rid of this task.

```
Module Sample =
        Begin
                Export DCGP, CGPC, CGPP
        End
Module Global_Gram (X : Sample) =
        Begin
                Module D = Respiratory_diagnosis
                Module T = Type_of_infection
                Module P = Previous_treatment
                Export Pneumococcus, Haemophillus, BGN
                Deductive knowledge =
                Rules:
                R001 If X/DCGP and D/Bacterial then
                        conclude pneumococcus is very_possible
                R002 If X/DCGP and D/Bacterian and T/Common_acquired then
                        conclude Pneumococcus is quite_possible
                R003 If X/CBGN and D/BCRO then
                        conclude Haemophillus is very_possible
                R004 If X/CBGN and D/BCRO and P/Previous then
                        conclude Haemophillus is quite_possible
                R005 If X/BGN and D/BCRO then
                        conclude BGN is very_few_possible
                        ...
                End deductive
        End

Module Sputum : Sample =
        Begin
                Import  Class_sputum
                Export DCGP, CGPC, CGPP
                Deductive knowledge
                Rules:
                    R001 If Class_Sputum = DCGP then
                              conclude DCGP is very_possible
                              ...
                    End deductive
          End
Module Sputum_Gram = Global_Gram(Sputum)
```

Figure 3. Example of generic module definition and application.

3. Execution Model of MILORD II

Deduction in MILORD was provided by classical inference engines (backward and forward) with uncertainty. This mechanism does not provide the possibility of complex communication as explained in section 1. We are implementing MILORD II with an inference engine based on partial evaluation. In this section we will explain the fundamentals of the partial evaluation in MILORD II and the general behaviour of the system.

3.1. Partial Evaluation

Partial evaluation takes as its input a program and a partial specification of the program input , and produces a new version of the program: a specialised program for the particular input values. Logic programming is specially suited for partial evaluation as shown in (VEN 84) and (GAL 86).

In the partial evaluation used in MILORD II a program is a partial KB or module, and the partial specification of the input of the program is a set of fact values. Each fact value of the input can specialise those rules of the KB which contain that fact in their premises. The specialisation of rules is performed by reducing the number of conditions of the rules, and obtaining new facts when conditions are exhausted. The new facts thus obtained can in their turn specialise other rules. The whole specialisation of the KB will finish when no rule of the KB has a known fact into its conditions, that is, when the KB is composed of a set of fact values and a set of rules without known facts in their premises.

The algorithm is based on two basic operations:

1) Reduction of the conditions:

Given a rule:

(A   B   …   Z∅ D) ρ = (A ∅ (B   …   Z ∅ D)) ρ where ρ is its truth value and a fact A with truth value α, we can partial evaluate the rule to obtain

$(B \quad \ldots \quad Z \varnothing D)\ \rho'$, where $\rho' = MP$[1] (α, ρ) is the new truth value of the rule.

<u>2) Modus Ponens:</u>

Given a rule,

$(A \varnothing B)\ \rho$

if we know that the truth value of fact A is α, we can deduce B with a certainty value $\beta = MP(\alpha, \rho)$.

3.2. Module Execution

In this section we will explain the general execution algorithm of a module. This algorithm is based on three elemental operations:

1) Questioning: This operation will obtain a question to make outside of the module (to the user or to another module).

2) The partial evaluator: This operation reduces the state of a module, taking into account the answer to the question obtained by the above mentioned operation.

3) Filter: This operation returns a set of formulas directly related to the goal of the module.

The execution of a module is initiated when a goal to prove by this module is set. From then on a loop starts between the operations of questioning and partial evaluation until the state of the module can not be further reduced, this will happen when all the conditions of the rules belonging to that state have unknown values. It is then that the filter operation starts and gives as result a subset of the formulas in the state directly related to the goal, i.e. the formulas in the deductive path to and from the goal in their reduced form.

4. Implementation

MILORD II is being implemented in two parts, the interpreter in Common Lisp and the compiler in C.

MILORD II is being implemented in Allegro Common Lisp for Macintosh machines. We use the window facilities of Allegro Common Lisp in order to present an easy tool for the users.

---

[1] MP (modus ponens) is a function that takes the truthvalue of the condition A, and the truthvalue of the rule ρ, and returns the truth value of the conclusion B.

5. Conclusions

Partial evaluation together with powerful modularisation techniques allow to define much structured and explanatory expert systems than with the classical approaches. The language MILORD II is the result of a research project who intends to adapt some techniques of wide use in software engineering to the languages of Artificial Intelligence. Concepts such Generic modules and refinement take a new meaning in the area of Artificial Intelligence that allow experts to express much more flexible their knowledge. Even more, it allows to verify the programming of KB's in an incremental way, this is a possible solution to one of the biggest bottle-necks in knowledge engineering: the validation and verification of KB's. The language is been used in the development of several expert systems, four of which are in the area of medicine.

References

(AGU 89) Agustí J., Sierra C., Sannella D., 1989 : "Adding generic modules to flat rule-based languages: A low cost approach", in Methodologies for Intelligent Systems, 4, Elsevier, pp. 43-51.

(BEL 91) Belmonte M., 1991: Renoir: Un sistema experto para la ayuda en el diagnostico de colagenosis y artropatias inflamatorias. Ph. D. Thesis, Universitat Autònoma de Barcelona.

(CHA 86) Chandrasekaran B., 1986 : "Generic Tasks in Knowledge-Based Reasoning: High-level Building Blocks for Expert Systems Design", Research Report, Ohio State University.

(GAL 86) Gallagher, J 1986: Transforming logic programming by specialising interpreters. Proceedings ECAI'86, pp. 109-122

(GOD 88) Godo L., López de Mántaras R., Sierra C., Verdaguer A. 1988: “Managing Linguistically Expressed Uncertainty in MILORD Application to Medical Diagnosis”. AI com., vol 1 n 1 pp 14-31.

(GOG 86) Goguen J. A., 1986 : "Reusing and Interconnecting Software Components", *IEEE Computer*, February 1986, pp. 16-28.

(HAR 89) Harper R., Sannella D., Tarlecki A., 1989 : "Structure and Representation in LCF", Proceedings of 4th IEEE Symp. on Logic of Computer Science.

(PLA 89) Plaza E., López de Mántaras R., 1989: "Model-Based knowledge acquisition for heuristic classification systems", *SIGART Newsletter*, April 1989, N 108, pp. 98-105.

(SIE 89) Sierra C., 1989 : MILORD: Arquitectura meta-nivell per a sistemes experts en classificació, Ph. D. Thesis, Universitat Politècnica de Catalunya.

(SIE 90) Sierra C., Agustí J. 1990 : COLAPSES: Towards a Methodology and a Language for Knowledge Engineering. Avignon'91 Tools, Techniques and Methods, pp. 407-423.

(VEN 84) Venken R. 1984: A Prolog meta-interpreter for partial evaluation and its application to source tranformation and query-optimisation. Proceedings ECAI'84, pp. 91-100.

(VER 89) Verdaguer A. 1989: Pneumon-IA: Aplicació dels sistemes experts al diagnòstic mèdic, Ph. D. Thesis, Universitat Autònoma de Barcelona.