Proceedings of the European Workshop on the Verification and Validation of Knowledge Based Systems

22-24 July 1991
Jesus College
Cambridge
England

Sponsored by Logica in association with the Department of Trade and Industry and the Commission of the European Communities
Verification by Construction in MILORD

C. Sierra, J. Agustí-Cullell, E. Plaza

Institut d'Investigació en Intel·ligència Artificial, CEAB-CSIC
Camí de Santa Bárbara
17300 Blanes, Catalunya, Spain.
Ph: 34 72 336101
Telex 56372 CEABL-E
Fax: 34 72 337806
Email: sierra@ceab.es @ceab.es plaza@ceab.es

ABSTRACT

Our approach to knowledge engineering is that verification during KBS construction is a necessary aspect for streamlining the KBS development process and for having a robust final system. The methodology proposed here stem from four basic components: a precise notion of partial KB or module, a set of module composition operations, an operation of module modification by refinement and a verification by construction procedures for these two operations. The process of design and development of a KB is represented with a Directed Acyclic Graph (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 creation and manipulation of this graph determines the MILORD language and the programming environment. The basic units of MILORD are modules and generic modules. The language provides three basic mechanisms of module manipulation: 1) refinement of modules, 2) composition by means of submodule declarations and 3) application of generic modules to other modules. Verification by construction involves a selection of certain development graph modification operations, the definition of the properties modifications should satisfy, and the implementation of verification procedures that check the satisfaction of those properties. Specifically, MILORD includes automatic verification of the operations of module refinement, module combination and generic module application.

Keywords: Formal approaches to KBS verification, Verification by construction, Knowledge Base verification, Modular Programming.

Research partially supported by Project SPES CICYT 880/382 and by PICS “Diseño e implementación de un entorno para la especificación y construcción de S. E.”
1. Introduction.

Inside second generation expert systems research, a knowledge level approach to problem solving is emerging [Chandrasekaran, 1986], [Steels, 1990], [Van de Velde, 1990]. Its goal is to model human problem solving expertise at a conceptual level, abstracting away implementation considerations. Concepts like generic tasks, methods or domain models have been successfully used to model expertise. Nonetheless these concepts are informal, their meaning becomes clear only by looking at their computational implementation. There have been few attempts to connect the informal knowledge level analysis of expertise and the implementation of the system [Van Harmelen and Akkermans, 1990]. We believe that formal specification methodologies share important features with knowledge engineering methodologies. Formal specification concepts like modules, genericity and refinement are notions close to those of knowledge level such as knowledge type, task, method, etc., being both techniques independent of the implementation. Nevertheless we will not address in this paper the comparison between the two. The goal of this paper is to present the language MILORD which could be used both as a formal specification language to formalise the knowledge level analysis and as an implementation language for KBS using a step wise refinement operation. Furthermore, we will show that this formal specification approach supports the automatic verification that certain KBS properties are satisfied during the construction of a knowledge base (KB), allowing the useful capabilities of incremental definition and modification of partial knowledge bases.

Section two discusses briefly some of the requirements a KBS language needs to fulfill with respect to the knowledge level formalisation, KBS development and verification. Next, in section three, we describe the methodology proposed and section four presents the language MILORD. Finally, section five explains the verification by construction operations in MILORD and we close with a some concluding remarks.

2. Some methodological requirements for knowledge engineering

Validation of KBS is a very complex issue, and put some demands upon the way a KBS is constructed and the structure that is built. A KBS is usually constructed incrementally, adding new information or changing some information in the KBS. Verification by construction provides the means to implement those transformations on a KBS and to check that some established specifications are complied by those modifications. Incrementality also entails the construction of a KBS is done by defining parts (or partial KB's) that are to be combined in a correct way to form the entire KBS. Thus verifying the composition of partial KB's into a coherent whole is a new requirement for the verification by construction approach.

A language that verifies by construction the incremental changes on a KBS, needs a set of elementary operations of transformation that are correct. Verifying the correctness of a change in a partial KB involves finding a chain of elementary transformation operations that link the original and the transformed partial KB. Correctness is verified only when such a chain can be found, and an incorrect change is detected if not found.

The methodology we propose here emphasise: modularity, incrementality, validation and multilanguage representation. Let us briefly discuss what is meant by this.

Modularity

The usual way of understanding a complex problem is to decompose it into simple subproblems which are combined using simple operations. To make a useful decomposition of problems, subproblems must have a simple and well defined interaction. In the framework of knowledge engineering modularisation can be seen as a formal structuring technique to organise the different types of knowledge that constitute a knowledge level analysis of problems. 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 knowledge engineering.

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.

Validation

The problem of KB validation has been tackled in other papers [López et al., 1989]. The validation techniques take into consideration mainly two problems: 1) Evaluation or statistical analysis of the behaviour of the system comparing it with the reality of its usage or with other experts (final adequacy) and 2) Syntactic verification of the knowledge base in order to detect and solve structural and logic errors (consistency). These properties have been applied only to the KB considered as a whole, without taking into account its building process. It is then necessary to think about these 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 [Harper, 1989]. An easy example of this is the use of different, local multi-valued logics in each module [Agustí et al., 1990].

We shall present a KB language which stems from the combination of three basic concepts: what is precisely a partial KB, which are the operations of composition of partial KB's and which is the refinement operation of a partial KB and we will show the opportunities they rise for verification during construction. We shall not deal here with the formalisation and operationalisation of the knowledge level concepts using MILORD. It is the topic of another paper [Agustí et al., 1990]. We shall not deal here either with control issues nor multilanguage representations. Nonetheless all these aspects are independent of the our three basic concepts and will be integrated in future versions of MILORD.

3. Proposed methodology

Our whole methodology revolves around the notion of module [Sannella and Tarlecki, 1989]. Modules formalise the previously mentioned notion of partial KB. The basic development unit of a large KB is the module, as opposed to other methodologies whose basic unit is the rule [Fiadiero et al., 1989], [Akkermans et al., 1989]. The modules are the partial KB's and consist of an input interface (set of facts) which shows how the module depends on the data obtained at run time, an output interface (set of facts and submodule identifiers) which tells what it brings to the rest of modules and to the user and finally a body which contains the definition of its behaviour. For instance, the module DM_5 in fig. 6 has no imported fact declaration, an exported fact: F, two submodules Ext_data, and B and a body containing some rule and meta-rule declarations.

The use of interfaces limits the interactions between modules defining precisely their possible outputs. In the first attempt to define a module the body can be left empty, that is, only the interface of the module has been decided. To fill the body of a module with behavioural contents incrementally is the key issue of the refinement methodology proposed in this paper.

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.

In the current implementation and in the proposed examples the basic language we have used is Milord, a rule-based language with a rule-based meta-language [Sierra, 1989]. In the future, this language will be extended with first order logic, ordered sorted types and a functional component.

3.1. Operations of composition

In this section we shall introduce the first approach to the module composition language of MILORD. The elementary operations of composition must allow either to build a module by joining already existing modules, or to divide the definition of a module into the definition of the set of its components. Both methods use the same concept of submodule. The body of the father module can use all exported facts of its submodules, but then the body of the submodules can not use any exported fact of the father module, to avoid circularities. For instance, in fig. 1 the module ext_data is seen to be defined from a group of modules which have not been defined yet (type1_data, ..., type_n_data). Ext_data is a module that collects information of a problem and divides the different sources of information in separate submodules. In the DAG straight lines represent the dependence of a module with regard to its submodules. In order to facilitate even more the reusability of modules, it is necessary to provide the user with the possibility of defining his own operations of composition. These can be represented by parametrised modules with respect to some of their submodules. Parametric modules are functions running from modules to modules. The parametric modules allow the generation of new modules only by their application to other existing modules. They can also be understood as a form of decomposition of the problem into an abstract -and thus reusable- part, and a concrete part. For instance, in figure 5 a generic module definition can be found. There, a module for making an analysis of gram is parametrised on the type of sample (it is taken from a medical application called Bacter-IA being developed in our laboratory). The common knowledge for all the analysis is contained in the body of the generic and the differences are captured in all the possible parameters. In that figure a possible parameter, Sputum, can be found.

3.2. Refinement operation

This operation captures some characteristics of incremental programming. It is a composition of the following three elementary operations: satisfiability of interfaces, hiding of interfaces and verification of the body's enrichment.

The satisfiability of interfaces guarantees that the refined modules will have the same exported set of facts than the module from which they have been refined, hiding, if necessary, the extra defined facts.

Hiding of interfaces is understood as the hiding of components of the output module interface, so that it will respect the output interface of the module from which it is a refinement, i.e. it makes the output interfaces equal.

The verification of the enrichment of bodies means that the refined module must have an enriched but still consistent set of rules from the father (some more rules are added).

In the graph of fig.3 these operations appear in bold arrows.

This operation allows us:
- To consider that all modules build up as a refined chain from a module have the same output interface.
- To check that a refinement chain is also an enrichment chain.
3.3. Graph of the KB

The graph of the KB symbolises the situation of the whole history of the development of the system. The closer to the root the more schematic design decisions, i.e. the top modules of the DAG can have defined only its interfaces.

The life cycle of a KB can be viewed as the application of some graph-modifying operations. Elemental examples of this are:

1. changes of implementation. A module containing a given submodule can change it by another one coming via refinements from a common ancestor. Different refinement chains means different implementation decisions.

2. changes of interface and hiding. Imposing a refinement relation between two modules in the graph means that the father imposes its interface to the son, using information hiding mechanisms.

3. refinements by means of generic modules. The module produced by the application of a generic module to a module parameter can be a refinement of the parameter.

The core of our methodological research is the definition of good graph operations and the combination of them into more complex ones. The result of this research will lead eventually to a graph manipulation language.

![Development Graph of a KB. An arrow means combination and a bold arrow means refinement.](image)

In the next section we introduce MILORD which can be used to handle this programming methodology.

4. The language MILORD

In this section the major elements of the language MILORD 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. Those modules whose contents are not defined in the paper will be written in italics.

4.1 Modules

The basic KB units written in MILORD 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 fig. 2).

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.

4.2 Primitive declarations

The most basic elements of MILORD are the same as in MILORD [Godo et al, 1987], [Sierra, 1989], the underlying current language of MILORD. These are facts of order 0+, production rules and meta-rules. Fig. 2 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: \texttt{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:

\texttt{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.

The next definition of interface will be used later in the definition of the refinement operation.

\texttt{Def The interface of a module is defined as}

\texttt{\texttt{int: Modexpr} \rightarrow \texttt{I x O x S}}

\texttt{where}

\texttt{Modexpr} \hspace{1cm} \text{is the set of module definitions}
\texttt{I} \hspace{1cm} \text{is the set of fact identifiers imported in the module.}
\texttt{O} \hspace{1cm} \text{is the set of fact identifiers exported by the module.}
\texttt{S} \hspace{1cm} \text{is the set of submodule identifiers.}

Kernel declarations

The kernel is made up of two components called \textit{deductive knowledge} and \textit{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. \textit{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 "$::$" (see fig. 6).
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, immunodepressed
Export ext_data1, ext_data2, ..., ext_datam
Deductive knowledge
Rules :
R1 if T1/X1 and immunodepressed
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

Ext_data

T1 ——> Tn

Type1_data Typen_data

T1 to Tn are labels in the archs standing for internal names of pointed modules inside ext_data

Figure 2. Module definition and its graphical representation.

In figure 2 the text of a module declaration can be found. Below it the graph representation of the module is presented. A graphic, and intuitive, representation of combinations between modules will be used along the paper.

4.3 Module and submodule declarations

Modules (or submodules) have the form:

Module modidentifier [ : modexpr1] [ = modexpr2]

where modexpr1 and modexpr2 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.
modexpr1 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 modidnominator in the environment where the declaration is made. In the next sections we detail these forms of module declarations.

Encapsulated declarations

The module Ext data (fig.2) 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.


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. For example,

```
Module A
Begin
  Module X=B
End
```

![Graphical representation of the declaration.](image)

Module Method of DA (fig 4) 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 unwanted 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 3 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.

4.4 Declaration of generic modules

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.
Module Domain\_model\_of\_DA =
Begin
    Module B : Sigext
    Export DA\_1; \ldots; DA\_n
End

Module Method\_of\_DA :
Domain\_model\_of\_DA =
Begin
    Module B = Ext\_data
    Module C
    Deductive knowledge =
        R\_1 \text{ if } \text{B/ext\_data, and } \ldots
        \text{ then } \text{DA\_1,}
        \ldots
        R\_m \text{ if } \text{B/ext\_data, and } \ldots
        \text{ then } \text{DA\_m,}
        \ldots
End deductive
End

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

Figure 4. Example of module refinement.

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 6. That example is obtained from Bacter-IA, a medical application being developed in our laboratory using MILORD. 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. [Agusti, Sierra, Sannella, 1989].

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, Haemophilus, 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 Haemophilus is very_possible
    R004 If X/CBGN and D/BCRO and P/Previous then conclude Haemophilus 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 5. Example of generic module definition and application.

4.5 Management of the KB development graph

The development cycle of a KB consists of the creation and manipulation of the graph of modules. The efficient manipulation of this graph should have to be carried out by a programming environment which includes visualisation and edition facilities. This environment is being designed, and some graph manipulations operations usual in KB's life cycle have been identified:

As examples of basic manipulations we have:

1) Implementation change. A new refinement of a module is provided, and some connections in the graph change.
2) **Refinement of the implementation of a submodule.** A submodule is substituted by a refinement of it.

3) **Refinement by means of a generic module.** A submodule is substituted by the application of a generic that produce a module refinement of the submodule.

The identification of new manipulation primitives integrated in the environment is part of our future research lines.

5. Verification operations

Currently MILORD offers an automatic verification of the for two kinds of operations: module composition and module refinement. In module refinement MILORD checks the interface declarations between a module and a refinement of it. In module composition MILORD checks that there exist a morphism between the logics of composite module and the logics of the submodule components. The existence of such a morphism assures that different logics can be combined coherently, that is to say assures that anything that can be deduced in a submodule component is also deducible in the composite module if is translated (via that morphism) from the logic of the first to the logic of the latter. We presently explain both verification operations.

5.1 Verification in module refinement

In module refinement MILORD checks the interface declarations between a module and a refinement of it. Refinement is a binary operation between modules made out of two elementary operations: consistency of enrichments and information hiding. The syntax of this operation is:

$$modexpr : modexpr$$

the syntax:

<table>
<thead>
<tr>
<th>Module modid : modexpr1 = modexpr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module modid = modexpr2 : modexpr1</td>
</tr>
</tbody>
</table>

The module `Method of DA` in figure 4 is a refinement of the module `Domain model of DA`. It states: (1) which facts `Method of DA` will export (i.e. `DA₁, DA₂, ..., DAₙ`) and (2) which of its submodules will be visible form the outside (i.e. `B`). It is so because they appear in the declaration of `Domain model of DA`. An example of a submodule hidden by the `:` operation can be seen between module `DM₁` and `DM_noexpert` in figure 6.

The `:` operation is defined as follows: `:` : $Modexpr \times Modexpr \rightarrow Modexpr \cup \{ error \}$

```plaintext
: (modexpr, modexpr') =
  If modexpr' = \lambda then return(modexpr)
  elsif enriches(modexpr, modexpr') then
    C = modexpr[S' := (S₁, S₂)] ∀S₁ S₂ :
    submodule(S₁, modexpr),
    submodule(S₂, modexpr'),
    identifier(S₁) = identifier(S₂)
    return(Hide(C, modexpr'))
  else return(error)
  endif
```

If the operation `:` affects two modules A : B the algorithm proceeds in the following way. First if B is null, i.e. we have a module declaration without refinement link (Module A = ...), the result will be A; if B is not null, we check if the module A is an enrichment of B (if it is not, an

---

1 $\lambda$ stands for the empty string.
error is raised). If the module A is an enrichment of B the operation is recurrently applied to the submodules of the modules applying the necessary hiding operations to preserve the interface of B.
The operations of enrichment and hiding will be explained in the sequel.

Enrichment

The enriches predicate checks the following properties:

(1) Preservation of interfaces

The ':' operation checks that a module A refinement of a module B preserves the interface of B.
Def Given A, B modules with int(A) = (I_A, O_A, S_A) and int(B) = (I_B, O_B, S_B) then
preserves(A,B) ≡ I_A ⊋ I_B ∧ O_A = O_B ∧ S_A ⊋ S_B
i.e. a module preserves the interface of another module if it keeps exactly the set of exported facts and does not eliminate any imported fact or any submodule of the interface to be preserved.

(2) Knowledge validation

Different properties of the structure of the knowledge contained in the modules are checked. Among them we emphasise the checking of the consistency of rules. Other criteria have been developed in the framework of the Esprit project VALID [López, Meseguer and Plaza, 1990] and will be included in future versions of the language.

(3) Logical properties

Modules contain a control component. Thus, a possible way of looking at a module is as the set of all possible rule sets that the control mechanism of the module can ever select. The operation ':' verifies that each rule set contained in a module has a monotonous extension in the refined modules.

Def rule_sets(M) = { R / R is a rule set compatible with the control of M }

Def We say that a module M is a monotonous extension of M', and write ME(M, M'), if ∀x ∈ rule_sets(M), ∃y ∈ rule_sets(M') : Th(x) ⊋ Th(y).

Understanding by Th(z) the deductive closure of the z rule set. The ⊋ operation in the context of the representation of uncertainty with linguistic terms (see [Godo et al, 1987] for details) has to be seen as:
Th(A) ⊋ Th(B) iff ∀x ∈ B, certainty(x, B) ≤_L certainty(x, A). with ≥_L the order relation in the set L of linguistic terms.

Hiding

The hiding operation Hide : Modexpr x Modexpr → Modexpr establishes the visibility of the submodules outside the module that contains them. It makes the visible submodules of a module A refinement of a module B to be the same as those of B. That is, we have the following definition.
Def Given A, B modules then
Hide(A, B) = A' :
A' equal to A except that ∀ S_i submodule of A' visible(S_i, A') = true iff visible(S_i, B) = true
Hide(A, nil) = A' :
A' equal to A except that ∀ S_i submodule of A' visible(S_i, A') = true
If a module is not a refinement of any other module (hide(A, nil)) then all its submodules are visible. This operation makes all the modules connected by a refinement use to have the same
set of visible submodules, exactly those of the root of the chains.

Module DM_1 =
  Begin
    Export F
  End
Module DM_Noexpert : DM_1 =
  Begin
    Inherit Ext_data
    Export F
  End
Module DM_2 : DM_Noexpert =
  Begin
    Inherit Ext_data
    Module B : P
    Export F
  End
Module DM_3 : DM_2 =
  Begin
    Inherit Ext_data
    Module B : P
    Export F
    Deductive knowledge
    R1 if B/H then F
    End deductive
  End
Module DM_4 : DM_3 =
  Begin
    Inherit Ext_data
    Module B : P
    Export F
    Deductive knowledge
    R1 if B/H then F
    End deductive
    Control knowledge
    MR1 if Ext_data ext dataj
    then inhibit-rules-using(B/H)
    End control
  End
Module DM_5 : DM_4 =
  Begin
    Inherit Ext_data
    Module B : P
    Export F
    Deductive knowledge
    R1 if B/H then F
    R2 if Ext_data ext data then F
    End deductive
    Control knowledge
    MR1 if Ext_data ext data
    then inhibit-rules-using(B/H)
    End control
  End

Figure 6. Refinement chain
In the modules of fig. 6 different examples of refinement can be seen. The refinements $DM_{\text{noexpert}}:DM_1$ and $DM_{\text{noexpert}}:DM_2$ stand for the incremental programming of the input interface. The refinement $DM_{\text{1}}:DM_{\text{2}}$ stand for a step of codification of the deductive component. The refinement $DM_{\text{3}}:DM_{\text{4}}$ is a coding of the control component. Finally the refinement $DM_{\text{4}}:DM_{\text{5}}$ makes an incremental refinement of the deductive component. A big part of the refinement steps need the reproduction of some pieces of code contained in preceding modules in the chain. This cumbersome task can be managed by the editor thus helping the user.

With the operations of submodule composition and refinement it is possible to verify some general properties of the structure of the graph. The most important property is the refinement transitivity.

5.2 Verification in module composition

In MILORD, a KB is the composition of partial KB’s called modules, specialised in having some knowledge for some specific (set of) issues. A module is conceptually formed by the declaration of a set of expressions (knowledge) in a local logic. Composite modules (and the whole KB) are constructed by composition of more elementary modules. In order to combine inferences derived by different logics in diverse modules, it is necessary to have a translation from the language of each subcomponent to the logic language of the composite module and that the translation satisfy certain properties. Essentially, what it is required from the translation is that to perform some inference in a local logic and afterwards translate it, is equivalent to first translate some formulae and perform the inference in the composite module logic. That is to say, the translation between modules involves no loss of deductive power.

As we will presently explain, given a composite module and translation mappings from each subcomponent module to the composite one, we are interested in verifying that the translation satisfies the inference-preserving properties stated above. In order to do so, we only have to prove that the translation mappings are morphisms between the involved logics.

More specifically, let’s assume $M$ and $M'$ be two modules and $(L, \vdash)$ and $(L', \vdash')$ their corresponding logics, $L$ and $L'$ standing for the languages and $\vdash$ and $\vdash'$ for the entailment relations defined on $L$ and $L'$ respectively. To establish a correspondence from module $M$ to module $M'$, a mapping $H: L \rightarrow L'$ relating their languages, is needed. In the following we will analyze some natural requirements to the mapping $H$ with respect to the entailment systems $\vdash$ and $\vdash'$. Henceforth $\Gamma$ and $e$ will denote a set of formulas and a formula of $L$ respectively.

R-1. If $\Gamma \vdash e$, then $H(\Gamma) \vdash' H(e)$

With this requirement we assure that for every formula deducible from a set of formulas $\Gamma$ in $M$, its correspondent formula in $M'$ by the mapping $H$ will also be deducible in $M'$ from the correspondent formulas of $\Gamma$ by $H$. In other words, there is no inferential power loss when translating from $M$ to $M'$ through a mapping $H$ satisfying R-1. Nevertheless the main drawback of requirement R-1 is that it does not forbid to deduce from $H(\Gamma)$, in $M'$, formulas that are not translations of any formula deducible from $\Gamma$ in $M$. These properties mean that, in the case of modules representing different experts, an expert $E'$ related to $M'$, using knowledge coming from an expert $E$ related to $M$, will be able to deduce the same facts than $E$, but not only.

R-2. If $H(\Gamma) \vdash' H(e)$, then $\Gamma \vdash e$

This is the inverse requirement of R-1. So, in this case all deductions in $M'$ involving only translated formulas from $M$ are translations of deductions in $M$, or equivalently if a fact is not deducible in $M$, then its correspondent fact in $M$ will not also be deducible from the translated knowledge.

R-3. If $H(\Gamma) \vdash' e'$, then there exists $e$ such that $\Gamma \vdash e$ and $H(e) \vdash' e'$.
This requirement assures that every formula deducible from $H(\Gamma)$ in $M'$ must in agreement with what can be deduced from $\Gamma$ in $M$. This requirement is slightly different from R-2, in the sense that it not necessary that $e'$ be exactly a translation of a deducible formulae from $\Gamma$, but only something deducible from such a translation. In the framework of logics for uncertainty management, $e'$ can be interpreted as a "weaker" form of $e$, i.e. a formula expressing more uncertainty than $e$.

It is worth noticing that, if $C$ denotes the consequence operator with respect to an entailment relation $|-\cdot$, that is, $C(\Gamma) = \{ e \mid \Gamma |-\cdot e \}$ for all set of formulas $\Gamma$, the requirements R-1 and R-2 can be rewritten in the following way:

R-1. $H( C(\Gamma) ) \subseteq C'( H(\Gamma) )$

R-2. $C'( H(\Gamma) ) \subseteq H( C(\Gamma) )$

being $C$ and $C'$ the consequence operators associated to the entailment relations $|-\cdot$ and $|-'$ respectively.

MILORD incorporates an algorithm that can check if for an application-dependent translation mapping between two modules satisfies that it is a morphism between the logics of those modules. This algorithm is explained in detail in an accompanying paper [Esteva et al. 91] and in [Agustí-Cullell et al. 91].

5.3 Verification in generic module application

In the definition of a generic module, some restrictions are imposed upon the admissible parameters (see Fig. 5). The restriction is a refinement relation between the admissible parameters and a given module. The application of a generic module involves checking the satisfaction of the refinement between the actual parameters and the modules from which they must be a refinement of. For instance, in figure 5, the Sputum_Gram module definition in made by the application of the generic module Global_Gram to the parameter module Sputum. The verification step consists of verifying that the module Sputum is a refinement of the module Sample. It can be seen that it is so because the export interface of sample is exactly the same as the export interface of Sputum.

6. Conclusion

We have analysed some methodological requirements of knowledge engineering and of verification based on our experience in developing KBS's. We have shown that the concepts of verification by construction, incremental programming and reflexive control can be based on a precise notion of partial knowledge base (module). We have designed a language for module composition and refinement. Generic modules enlarge the reusability concept, allowing the construction of large knowledge bases from a library of reusable generic modules. The cycle of a KB development has been represented by means of a DAG where nodes stand for modules and arcs for operations. The life cycle of this DAG entails the application of some graph-modifying operations. The definition of powerful graph modification operations from more elementary ones is the core of the methodology of our future research. As a general conclusion, verification by construction involves a selection of certain development graph modification operations, the definition of the properties modifications should satisfy, and the implementation of verification procedures that check the satisfaction of those properties. Specifically, MILORD includes automatic verification of the module refinement and combination operations.
References
López B., Meseguer P., Plaza E., 1989: An state of the art on KBS validation, AI Communications, 3(2) 58-72.
Van Harmelen F., Akkermans H., 1990: Formal Specifications of Knowledge Models, ESPRIT Document RFL/ECN/L4/1