AVIGNON '91

The texts of the papers given at the Eleventh International Avignon Conference, together with the description of the products presented at the exhibition, are published in the following volumes.

Les textes des conférences données lors des Onzièmes Journées Internationales d'Avignon ainsi que les fiches descriptives des produits présentés à l'exposition sont regroupés dans les volumes suivants:

General Conferences
Conférences Générales

- Volume 1 • Tools, Techniques & Methods
- Volume 2 • Second Generation Expert Systems
- Volume 3 • Applications

Specialized Conferences
Conférences Sectorielles

- Volume 4 • Artificial Intelligence, Defense & Civilian Security
- Volume 5 • Artificial Intelligence, Telecommunications & Computer Systems
- Volume 6 • Artificial Intelligence & Transportation
- Volume 7 • Artificial Intelligence & Electrical Engineering
- Volume 8 • Natural Language Processing & its Applications
- Volume 9 • Applications

Exhibition Catalog (1 volume)
Catalogue de l'Exposition (1 volume)
COLAPSES : TOWARDS A METHODOLOGY
AND A LANGUAGE FOR KNOWLEDGE ENGINEERING

Carlos Sierra & Jaime Agustí-Cullell

CEAB
Cami de Santa Bàrbara
17300 Blanes, Girona
Spain

Tel. : +34 72 336101

Abstract : Unlike in software engineering (SE) where the problem specification is the starting point, the goal of knowledge engineering (KE) is to obtain an adequate specification of a problem, that is, an adequate knowledge base (KB). Thus, the methodology of KE must bear the heavy process of creating and manipulating an evolving space of KB's, which becomes increasingly adequate. The methodology proposed here stems from three basic components: a precise notion of a partial KB or module, a set of module composition operations, and a notion of module modification by refinement. 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 COLAPSES language and the programming environment. The basic units of COLAPSES 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. Some elements of the language semantics and execution model are provided.

Keywords : knowledge base development, software reusability, modular programming, knowledge base refinement.

Research partially supported by Project SPES CICYT 8801282 and by PICS - Diseño e implementación de un entorno para la especificación y construcción de S. E.
1. INTRODUCTION

Inside second generation KBS research a Knowledge Level theory of problem solving is emerging, [Chandrasekaran, 1986], [Steels, 1989], [Van de Weide, 1990]. Its goal is to model human problem solving expertise at a conceptual level abstracting from implementation considerations. Concepts like Generic Task, Method or Domain Model 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., both techniques being 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 Collapses which could be used both as a formal specification language to formalize the knowledge level analysis and as an implementation language for KBS using a step wise refinement operation. This language is an attempt to extend a previous knowledge based language called Milord [Godo et al., 1987], [Sierra, 1989], used to develop several medical expert systems [Verdaguer, 1989], with some of these formal specification concepts.

Section two discusses briefly some of the requirements this language needs to fulfill with respect to the knowledge level formalisation and KBS development. Next, in section three, we describe the methodology proposed and finally, section four presents the language Collapses.

2. SOME METHODOLOGICAL REQUIREMENTS FOR KNOWLEDGE ENGINEERING (KE)

Experience in KB design, specially using knowledge acquisition techniques [Plaza, Lopez de Mantaras, 1989], 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 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, modularization 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 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 [Chandrasekaran, 1986] [Goguen, 1986]. For instance, although the diagnosis of infectious chest illness and that of chest tumors 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.

These generic units can express the generic types of knowledge that constitute the recurrent components of problem solving that are generic across different domains. For instance, different concepts of the knowledge level like generic tasks or methods can be formalised using generic modules. A method will be understood as a generic module that takes a task (module) as a parameter and produces a new module containing a hierarchy of submodules (subtasks) and a control component on them.

Incremental modification of KB’s

The KB building methodology is an iterating two-step process. First a prototype is built (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. So it must be a component of each partial KB. This locality of control allows us to tailor the control to the concrete subproblem being solved. A declarative way of defining the control is preferable using meta-level control declarations and reflection mechanisms.

Validation

The problem of KB validation has been tackled in other papers [Lopez et al., 1989]. 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 validation of the knowledge base in order to detect and solve structural and logical 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 whose, 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 multi-valued logics in each module [Agusti 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. We shall not deal here with the formalisation and operationalization of the knowledge level concepts using Collapses. It is the topic of another paper [Agusti et al., 1990]. Neither shall we deal here with control issues or validation of partial KB’s or multilanguage representations. Nonetheless all these aspects are independent of the above mentioned three basic concepts and will be integrated in future versions of Collapses.
3. PROPOSED METHODOLOGY

Our whole methodology revolves around the notion of module [Sannella and Tarlecki, 1989]. Modules formalize 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 [Fidore 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. 5 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 by 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 paragraph we shall introduce the first approach to the module composition language of COLLAPSEs.

The elementary operations of composition must make it possible 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 cannot 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 (type f_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 parameterized 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 6 a generic module definition can be found. There, a module for making an analysis of gram is parameterized 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 as the modules 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 built 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.
4. PROPOSED LANGUAGE: COLAPSES

In this section the major elements of the language COLAPSES are presented by means of an example. Most 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 COLAPSES 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 hierarchical 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 module definitions.

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 COLAPSES are the same as in MILORD [Godo et al, 1987], [Sierra, 1989], the underlying current language of COLAPSES. These are facts of order 0, production rules and meta-rules. Fig. 2 shows the primitive declarations outlined in this section.
4.3 Module and submodule declarations

Modules (or submodules) have the form:

\[ \text{Module } \text{modidentifier}[\cdot \text{modexp}] = \text{modexp}. \]

where \( \text{modexp} \) and \( \text{modexp}_r \) 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.

\( \text{modexpr} \) defines a module from which the module \text{modidentifier} is a refinement.

The symbol \( . \cdot \) stands for the module refinement operation and the symbol \( \rightsquigarrow \) for the module composition operation. If \( \text{modexpr} \) and \( \text{modexpr}_r \) are empty strings, the effect of the declaration is just to keep the \text{modidentifier} in the environment where the declaration is made. In the next sections we detail these forms of module declarations.

Encapsulated declarations

The module \text{Ex_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.

Declarations by reference : module composition

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

\[ \text{Module A Begin Module X=B End} \]

Graphical representation of the declaration.

Module \text{Method_of_DA} (fig 4) references module \text{Ex_data} naming it B. This declaration makes all facts exported by \text{Ext_data} be visible in the kernel of \text{Method_of_DA}. Thus, \text{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 \( \cdot \cdot \). The concatenation of prefixes allows the path to a fact to be represented 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 \text{Inherit} (see module \text{OM_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 \( . \cdot \) operation allows it, i.e. if the submodule is not hidden by \( \cdot \cdot \). For instance in figure 3 the submodule \text{Csigxt} is not visible outside \text{method_of_DA} because \text{Domain_model_of_DA}
does not contain any submodule called C. So, Method_of_DA/C/P cannot 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**

```plaintext
Begin
Module B : Sigext
End
Begin
Module C : Ext_data
End
Deductive knowledge = R, if B/ext_data and ...
then DA,
... if B/ext_data and ...
then DA,
... Graphical representation of the modules and example of verification step over the graph structure of the modules.
End deductive
End
```

**Figure 4. Example of module refinement**

**Application 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.

**4.4 Module refinement**

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 modexpr1 modid : modexpr2 = modexpr1.

The syntax is equivalent to: modid = modexpr1, modexpr2 = modexpr1.

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. DA, DA, ..., DA) and (2) which of its submodules will be visible from 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 DM_1 and DM_noexport in figure 5.

The "" operation is defined as follows: modexpr1 modid : modexpr2 = modexpr1 ∪ [error] = (modexpr1, modexpr2).

If modexpr1 = \{U \} then return(modexpr1)
else enriches(modexpr1, modexpr2) then return(Hide(C, modexpr1))
else return(\{error\})

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 error is raised). If the module A is an enrichment of B the operation is recursively 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 following.

**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 Esprid 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).

\( \lambda \) stands for the empty string.
Understanding by Th(Z) the deductive closure of the rule set. The $\supseteq$ operation in the context of the representation of uncertainty with linguistic terms (see Godo et al., 1987) for details has to be seen as:

$$\text{Th}(A) \supseteq \text{Th}(B) \text{ iff } \forall x \in B, \text{ certainty}(x, B) \leq \text{ certainty}(x, A).$$

with $\leq$ the order relation in the set L of linguistic terms.

Hiding

The hiding operation $\text{Hide}(\text{Modexpr} \times \text{Modexpr}) \rightarrow \text{Modexpr}$ 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 the same as those of B. That is, we have the following definition.

Def Given $A, B$ modules then

$$\text{Hide}(A, B) = A' :$$

A' equal to A except that $\forall S$, submodule of $A'$ visible($S$, A') = true iff visible($S$, B) = true

$$\text{Hide}(A, \text{nill}) = A' :$$

A' equal to A except that $\forall S$, submodule of $A'$ visible($S$, A') = true

If a module is not a refinement of any other module (hide(A, nill)) then all its submodules are visible. This operation makes all the modules connected by a refinement tree have the same set of visible submodules, exactly those of the root of the chains.

In the modules of Figure 5 different examples of refinement can be seen. The refinements $\text{DM}_1$, $\text{DM}_2$, $\text{DM}_3$, $\text{DM}_4$, and $\text{DM}_5$ Noexpert stand for the incremental programming of the input interface. The refinement $\text{DM}_3$ stands for a step of codification of the deductive component. The refinement $\text{DM}_4$ stands for a step of codification of the control component. Finally, the refinement $\text{DM}_5$ makes an incremental refinement of the deductive component. A big part of the refinement process needs 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.

<table>
<thead>
<tr>
<th>Module DM_1 = Begin</th>
<th>Export F</th>
</tr>
</thead>
<tbody>
<tr>
<td>End</td>
<td></td>
</tr>
</tbody>
</table>

| 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 |

4.5 Declaration of generic modules

The definition of generic modules offers 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 shown in figure 5. That example is obtained from BacterIA, 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. 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 a 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 automated by the compiler, so that the user gets rid of this task.

```plaintext
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/Bacterial and T/Complex_acquired then
                    conclude Pneumococcus is quite_possible
                R003 If X/CBGN and D/BCRO then

    End
End module
```

Figure 6. Example of generic module definition and application.

4.6 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 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 produces a module refinement of the submodule.

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

5. Semantics : compilation and execution model

The semantics of the language has been defined in a denotational style. The most important semantic functions are generation and linkage, which carry out the compilation of the abstract syntax into the semantic objects.

The generation operation creates the code associated to a concrete module by marking its dependence of other modules, both of composition and of refinement. The linkage operation receives pairs of codes associated to modules and tries to solve the interdepen-
encies by generating a new code structure. These operations act as coroutines in the compilation of the module hierarchy.

The construction of the internal representation of the graph through semantic objects has new aspects to take into consideration. Usually the identifier environments of compilers were built in a stack way by adding new bindings on top of it when going in the hierarchical structure and erasing bindings when going out [Agusti, Sierra, Sannella, 1989; Hasselglen, 1989]. This did not allow references to components which had not been previously defined (in one step compilers, as ours). Our compiler eliminates this restriction and allows a separate compilation. Moreover, the semantics allow a concurrent compilation.

More details for the compilation process can be found in the report [Sierra, Agusti, 1990]. Let's now consider some aspects of the execution model of the modules.

The execution model of a module is based on the filtering of the production rules and of the submodules by the meta-control component. This meta-control component is able to eliminate some submodules and some rules in a module. This is done in order to be able to use the module to create execution needs. Meta-control works by a mechanism of implicit reflection [Maes, Nardi, 1988]. It is activated when a new information is obtained, either from the exterior or as an outcome of the deductive component of the module, or as a predicate exported by a submodule. The execution of a module is the execution of its deductive behaviour following the order established by the meta-control. Besides the reflection mechanism mentioned above, this execution is determined by the evaluation type:

Lazy evaluation: determines that the evaluation of the elements in the export interface is to be done only when needed. The rules will be executed goal-driven, and the expected facts of submodules will be evaluated only when the rules ask for them.

Eager evaluation: first of all, all the submodules are executed trying to get answers for all its export interfaces. After that, all the import interface of the module will be asked to the user, and finally all the rules are data-driven executed.

Again, this evaluation parameter is local to each module, and affects only its execution. In a hierarchy of modules different evaluation policies can be used at different levels. This execution model is heavily based on the characteristics of MILORD, the base language of COLAPSE.

6. CONCLUSION

We have analyzed some methodological requirements of KE based on our experience in developing KB's. We have shown that the concepts of validity, reusability, incremental programming and reflexive control must 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. The efficient manipulation of these graphs would have to be carried out by a programming environment which includes visualization and edition facilities.

References