Mathware & Soft Computing 4 (1997) 299-338

Milord II: Language Description

Josep Puyol-Gruart, Carles Sierra
Artificial Intelligence Research Institute (IIIA)
Spanish Scientific Research Council (CSIC)
Campus UAB. 08193 Bellaterra, Catalonia, Spain
e–mail: {puyol,sierra}@iiia.csic.es

Abstract

In this paper we describe the language **Milord II**. The description is made in terms of computer language concepts and not in terms of the logical semantics underlying it. In this sense the paper complements others in which the focus of the description has been either the object level multi-valued language description, or the reflective component of the architecture, or even the several applications built using it. All the necessary elements to understand how a system programmed in **Milord II** executes have room in this full description: types, facts, rules, modules, local logics, control strategies, ... Although the description is guided by the **Milord II** language syntax, this is by no means a user's manual, which would deserve a much longer document, but a language summary description that places all the components of the language in their correct place.
Keywords: Expert Systems, Knowledge-Based Systems, languages, uncertainty.

1 Introduction

This paper summarises the outcome of the research developed in our laboratory during several years, in the form of a computational language. In this sense the merit is, if any, not only on the authors but on all the members of the laboratory. The focus of the paper content is the description of the language structure and its constructs, but not its logical semantics. In [22] you can find a complete description of the logical semantics underlying **Milord II**. The interested reader will find in the bibliography several references to **Milord II** papers that will help in the deepening of the theoretical grounding that is only mentioned here. This paper must not be understood as a user's manual but as a concise description of the language. The complete description of all the possibilities of usage, the environment facilities, and complex examples would deserve a much longer document.

When one focus on the job of writing down the definition of a language it is always difficult to decide the starting point. Which construct is the most primitive? Which explanation should go first to make the understanding of the rest easier?

Any construct of the **Milord II** language, as of any other modular language, is difficult to grasp without, at least, a superficial view of what the modular system is. Most constructs obtain a full meaning when placed in the context of a module. Nonetheless, a module is just a collection of those constructs. To avoid this circularity we begin with the introduction of the basic modular system. Then we go through all the constructs in the language, from the elementary facts to the meta-rules and the generic modules. In this walk through the language we will be guided by the syntax. To do so we rely on a syntactic definition a-la-BNF in which the symbols ::=, [,], |, $^+$ are part of the BNF formalism, as follows:

L ::= R	The syntax of L is defined by R
[X]	An optional item
X | Y	An item from one of the syntactic categories X or Y
X^+	One or more occurrences of the syntactic category X

- We write the predefined terminal symbols that are part of the language **Milord II** in **underlined boldface**.
- We write user-defined terminal symbols in *italic*. They are always atomic symbols or strings.
- We write non-terminal symbols in normal type face.

Comparison of symbols is case-insensitive. Lines of comments can be written after two semicolons (;;). If the comment is larger that one line, two semicolons must be written at the beginning of each line. Several spaces and carry returns are ignored and considered only a space.

2 Modules

The most primitive structural construct of **Milord II** is the module. A program (see Figure 1) is composed by a set of module declarations that can recursively contain other module declarations, then forming a hierarchy. Module declarations are surrounded by the keywords `begin end`. Between these two keywords all the components of the module, including submodules, must be declared; nothing belonging to a module can be declared outside these two keywords. Module declarations can be given a name, that becomes its identifier, see `amodid` in `moddecl`. The scope of these identifiers is the module declaration text –including the submodule declarations textually declared inside the module. Then, to refer to a submodule component (fact, rules, ...) we must prefix the identifier of the component by a path of the module identifiers of modules placed syntactically between the point of reference and the point of definition (`pathid`). Components in a path are separated by "/". Top level modules are then not prefixed.

Local names can be given to previously defined modules by allowing a reference to their name in the right part of a module name binding, see `pathid` in `bodyexpr`.

The contents of a module declaration can be clustered in the next sets of declarations: hierarchy, interface, deductive knowledge and control knowledge. The use

PROGRAM	::= moddecl$^+$
moddecl	::= **Module** *amodid* [≡ bodyexpr]
bodyexpr	::= **begin** decl **end** | pathid
pathid	::= *amodid* | *amodid*/pathid
decl	::= [hierarchy] [interface] [deductive] [control]
hierarchy	::= moddecl | hierarchy hierarchy
interface	::= [**Import** predicateidlist] [**Export** predicateidlist]
predicateidlist	::= *predid* , predicateidlist | *predid*

Figure 1: Syntax of module definition.

of empty module declarations (notice that `bodyexpr` is optional) will be explained in Section 7.

Hierarchy: Is a set of module declarations. We say that these modules are submodules of the module that contains them.

Interface: It has two components, the import and the export interface. They declare which facts could be asked to the user[1] (import) and those that can be results of the module (export). All the facts inside a module not declared in the export interface are hidden to the outside of the module.

Deductive and Control: These declarations allow modules to compute the components of its export interface (output) from the components of its import interface and those of the export interfaces of its submodules (input). Deductive knowledge includes the declarations of the object language which in our current implementation is mainly a rule-based language. Control knowledge is declared by means of a meta-language which acts by reflection over the deductive knowledge and the hierarchy of submodules. A module with an empty deductive and control components is considered to be a pure interface (a signature).

In the subsequent sections we will describe in detail the different components that can be included inside a module definition. We will end up with a detailed description of the modular system of **Milord II**.

3 Deductive Knowledge

The deductive knowledge component of a module is composed by a dictionary declaration, that is, the set of facts (those belonging to the interface of the module and other intermediate facts) and their attributes (see Figure 2); a set of rules declaration, that is, a set of propositional weighted rules; and an inference system declaration, that is, the local logic of the module.

[1] In some cases the control of the module can give a value to an imported fact before asking it to the user (see Section 6.4).

deductive	::= **Deductive knowledge**
	[**Dictionary:** [**Types:** typebinding$^+$] [**Predicates:** predicate$^+$]
	[**Rules:** rule$^+$]
	[**Inference system:** logcomp]
	end deductive
typebinding	::= *typeid* [= typespec]

Figure 2: Syntax of deductive knowledge definition.

The dictionary is composed of an optional type definition (see Section 3.1.1) and the declarations of the facts and their attributes.

3.1 Facts

Facts are the simplest knowledge representation unit in **Milord II**. They are named structures that represent the concepts used in a module. The declaration of a fact is made by binding an atomic name, that is, the identifier of the fact (`predid`), with a set of attributes (see Figure 3).

predicate	::= *predid* = attributes
attributes	::= [name] [question] type [function] [relation$^+$] [explanation] [image]
name	::= **Name:** *string*
question	::= **Question:** *string*
type	::= **Type:** typespec
function	::= **Function:** (S-expression)
relation	::= **Relation:** relationid pathpredid
pathpredid	::= pathid/*predid* | *predid*
relationid	::= **Needs** | **Needs_true** | **Needs_false** | **Needs_value** |
	Belongs_to | **Needs_quantitative** | **Needs_qualitative** | *symbol*
explanation	::= **Explanation:** *string*
image	::= **Image:** *fileid*

Figure 3: Syntax of fact definition.

3.1.1 Types of facts

The type is the only attribute that is mandatory in a fact declaration. It determines the set of values a fact can take. For instance the fact *dead* would be a boolean predicate (it is false or true); and the facts *temperature* and *voltage* should be of numeric type.

A fact is valuated over the set of values determined by its type plus the special value *unknown*, meaning ignorance of the value. There are four basic predefined types, namely *boolean*, *many-valued*, *numeric* and *class*; and a parametric user-defined type named *fuzzy*. Moreover, programmers can declare two anonymous

typespec	::= **boolean** | **many-valued** | **numeric** | **class** | **fuzzy** char-funct | (symbollist) | (valuesspec) | *typeid*
symbollist	::= *symbol* [*string*] | symbollist , symbollist
valuesspec	::= *symbol* [*string*] char-funct | valuesspec , valuesspec
char-funct	::= (*number* , *number* , *number* , *number*)

Figure 4: Syntax of type definition.

types by enumerating the values that the fact can receive, namely *set* and *linguistic*. Next there is a summary of the meaning of **Milord II** fact types:

Boolean Facts These are facts whose value can be either *Yes* (*true*) or *No* (*false*).

Numeric Facts The value of a fact of this type is a real number. They are used to represent quantitative data, for instance concepts like *temperature*, *number of leucocytes*, etc.

Many-valued Facts The concepts represented as facts with many-valued type are those whose truthness is graded. For instance if we use a subjective criteria to appreciate if a patient has fever by touching him with the hand, we can consider that the fact *fever* is a many-valued fact (we can say that *fever is possible*). In this case that fact is declared as

```
fever = Type: many-valued
```

The type many-valued is parametric with respect to a set of linguistic terms, representing truth values, defined by the programmer. This set of terms must be defined in the inference system declaration (see Section 4). The value of facts of this type will be an interval over the so defined ordered set of linguistic terms.

Class Facts This type is a bit special. The set of values of this type is empty. Hence, facts of this type will be valueless facts. As a direct consequence of this, facts of this type cannot appear in premises or conclusions of rules. They can be used in the structuring of knowledge as relations between facts. These relations may appear in premises of meta-rules. So, for instance, we can declare the fact *oral* as a class fact. Then, we can define relations between the antibiotics that are administrated orally and the fact *oral*. Finally, we could define a meta-rule to be applied to all antibiotics administrated orally by using the previously defined relation in the meta-rule's premise.

Fuzzy Facts Vagueness of concepts as *fever* can be interpreted as the degree of membership of a numeric measure (in this case *temperature*) to a fuzzy set.

The values of facts with an associated fuzzy set are still intervals of linguistic terms. The way of computing the interval will be done, in this case, by the

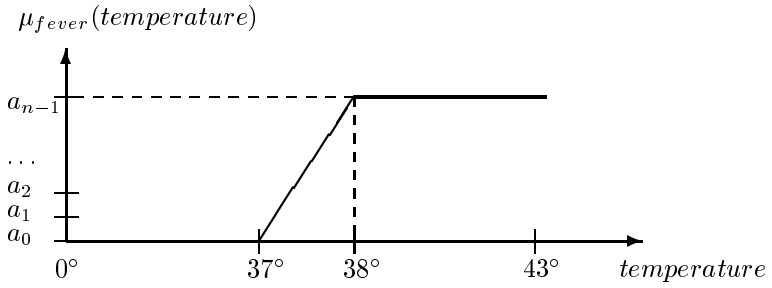

Figure 5: Fuzzy set representing the concept *fever*.

application of the fuzzy membership function to the numerical value of the fact that must appear in a relation named *needs_quantitative*.

We can see an example of fact declaration of the concept *fever* (see Figure 5). This concept is declared by giving the four points of the trapezoidal approximation of its membership function.

```
fever = Type: fuzzy (37,38,43,43)
        Relation: needs_quantitative temperature
```

Attributes of relations will be explained in Section 3.1.4.

Set Facts Facts of this type get values from a user-defined finite set of symbolic values. This set is defined by enumerating its elements. For instance, the fact *treatment* gets values from a set of *antibiotics* (*etambutol*, *aciclovir* and *ganciclovir*). This set is, moreover, the anonymous type of the fact *treatment*.

```
treatment = Type: (etambutol, aciclovir, ganciclovir)
```

The value of the fact *treatment* will be a mapping from the elements of its type to intervals of linguistic terms representing the degree of membership of every *antibiotic* to the fact *treatment* (v.g. $\mu_{treatment}(etambutol) = [\alpha, \beta]$).

Linguistic Facts We can declare a linguistic fact by giving a user-defined set of linguistic values and giving for every linguistic value a trapezoidal approximation of a fuzzy set with respect to a numeric fact. In Figure 6 we can see a new representation of the fact *fever* by means of three fuzzy sets, that is, *low*, *medium* and *high*.

The declaration of this new interpretation of the *fever* concept can be:

```
fever = Type: (l "low"    (37,37.3,37.6,38),
               m "medium" (37.6,38,38.5,39),
               h "high"   (38.5,39,43,43))
        Relation: needs_quantitative temperature
```

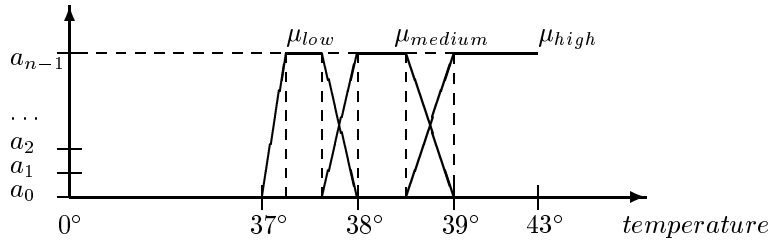

Figure 6: Fuzzy sets representing the concept *fever*.

Notice that, as in the case of fuzzy facts, it is necessary to declare the same relation *needs_quantitative* with a numeric fact, in this case again *temperature*. The optional string in the elements of the type (e.g. `low`) is used only for informational purposes.

3.1.2 Fact Functions

Facts may have a functional expression attached to them. This is the way we incorporate functional programming into **Milord II**. A fact with such an expression attached to it will get its value as the result of the evaluation of the expression.

Possible applications of fact functions could be: interfaces with other programs (windows, networks, statistics, etc), or procedural computations.

S-expression	::= *atom* | list | predef-func | S-expression S-expression
list	::= (S-expression) | ( )
predef-func	::= (**Type** *predid*) | (**Linguistic_terms**)

Figure 7: Syntax of function definition.

The function attribute (see Figure 7) of a fact is programmed in Common Lisp[2] with some extensions, it is a *S-expression*. This *S-expression* is considered to be evaluated inside an environment that contains variables, with the same name of the facts declared into a module, bound to the fact's value. It means that any fact name appearing in a *S-expression* will be considered as a local variable and its evaluation will return the current fact value.

The meaning of the two **Milord II** predefined functions is:

Type: This function applied to a fact name of the current module returns the type of that fact in the following form: for boolean, many-valued, numeric and class types, the symbols `boolean`, `many-valued`, `numeric` and `class` respectively; for fuzzy types, a list containing the keyword `fuzzy` and the membership

[2] As the underlying language of **Milord II** is Common Lisp, then it is easy to use code programmed in this language. In this Section we will use Common Lisp terminology, as we think it is well known. Details of this can be found in [25].

function in list form, for instance `(fuzzy (37 38 43 43))`; similarly for set and linguistic types, a list containing the parameters used to declare the concrete type, for instance for a linguistic type `((l "low" (37 37.3 37.6 38)) (m "medium" (37.6 38 38.5 39)) (h "high" (38.5 39 43 43)))`

Linguistic_terms: This function returns the set of linguistic terms of the current module in list form.

It is necessary to know the internal representation of the values of facts depending on their type. This is useful to manipulate the values of facts in a function attribute and to know which is the value format that a function must return depending on the type of the fact that contains it. We remind you that a fact with a functional expression attached to it will have the result of the evaluation of such expression as value.

Boolean: It is a list of two elements: `(false_etiq, false_etiq)` for *No*; and `(true_etiq, true_etiq)` for *Yes*. `false_etiq` and `true_etiq` are respectively the first and the last element of the list of linguistic terms declared in the local logic of the module.

Numeric: It is a real number.

Many-valued and Fuzzy: It is a list of two elements `(etiq_1, etiq_2)` where `etiq_1` and `etiq_2` are elements of the list of linguistic terms and $etiq\_1 \preceq etiq\_2$; where $\preceq$ is the total order defined implicitly when declaring the linguistic terms.

Set and Linguistic: It is a list containing lists of two elements: the first is an atom representing the name of an element of a set fact or the linguistic value of a linguistic fact; and the second is a pair representing an interval of linguistic terms as in many-valued and fuzzy types. For instance for the fact *fever*: `((low (false false)) (medium (possible true)) (hight (possible true)))`. This list contains the elements in the same order as declared in the type attribute.

Now we can see an example of function declaration of the *Terap–IA* [6] application[3]. The value of the *clearance_of_creatinina* can be obtained by means of the following expression.

$$clearance\_of\_creatinina = 140 - \frac{age \times weight \times \left\{ \begin{array}{ll} 1.0 & \text{if } male \\ 0.8 & \text{if } female \end{array} \right\}}{72}$$

We can see the declaration of the fact *creat_clear* that contains a function attribute to compute its value.

[3] An application for pneumonia treatment.

```
Creat_clear= Name: "Clearance of creatinina"
             Type: numeric
             Function: (let* ((true (first (linguistic_terms)))
                              (male (equal (second (first sex))
                                           (list true true))))
                         (- 140 (/ (* age weight (if male 1.0 0.8))
                                   72)))
```

In this function `age` and `weight` are numeric facts, and `sex` is a set fact of type `(male, female)`.

3.1.3 Interface Attributes

There is a set of attributes that customise the interface associated to a particular fact. The attributes are:

Name: This is an attribute that associates a long name, i.e. a string, to the fact. Experts use atomic short mnemonic names to identify a fact mainly to simplify rules and meta-rules writing. The long name is the representation of the fact to be shown to users to help them understand the concept the fact represents.

Question: The value of this attribute is a string that the system uses to query a user about the value of the fact.

Explanation: When the system asks the user for the value of a fact, the user can ask for help. In that case the explanation attribute value, a string, is presented to the user.

Image: It is the name of a file containing an image to be shown together with the question when that fact is asked to the user. For instance in *Sponge–IA* application[4], when the system asks about the external aspect of the sponge the user wants to classify, an image containing pictures representing the possible forms is shown to him.

3.1.4 Fact Relations

Establishing relations between facts is a very useful and common mechanism of knowledge structuring. Relations are declared as fact attributes. A fact can have several of these relation attributes. These attributes define named relations between the fact with the attribute and other facts of the system. By means of these declarations a labelled directed graph of facts is established. This graph can be used to define a hierarchy of facts, to establish an order of evaluation, etc. We can distinguish between relations defined by the programmer and predefined relations. The programmer can define relations among facts and give them sense and properties when defining the control component of the module because these relations can appear as conditions of meta-rules, Section 6. Here we present some examples of relations defined by the medical expert programming the *Terap–IA* application:

[4] An application in the classification on marine sponges [10]

- *doxicicline equivalent_spectrum roxitromicine* : The facts *doxicicline* and *roxitromicine* are antibiotics. This relation means that both antibiotics can treat the same kind of pneumonia.
- *amoxiciline belongs_to oral_administration* : The antibiotic *amoxiciline* belongs to the group of the antibiotics administrated orally.

There is a set of common problems faced by experts that can be solved using relations. Predefined relations are relations with a system-defined meaning.

Belongs_to: Experts usually build hierarchies in their applications. This predefined relation, meaning inclusion, is thought for that purpose and makes an off-line application of the transitive property of inclusion, giving a more efficient system. For instance when the expert defines the next two relations between antibiotics, *doxi belongs_to tetras*, and *tetras belongs_to tetraciclines*, the system adds the new relation *doxi belongs_to tetraciclines*.

There are several relations which are used to induce the order of the questions to be made to the user when getting values for facts in the import interface.

Needs: When the system is going to ask for the value of a fact, and this fact has a *needs* relation with another fact, the last fact will be asked first. For instance the relation *pregnant needs sex* means that before asking if a person is *pregnant* the system will ask for the value of fact *sex*. This relation can be used between facts of any type.

Needs_true and Needs_false: This cases are similar to the previous one, but the behaviour is different depending on the answer to the first question being asked. Consider the following example of relation, *pregnant needs_true female*. The first question to the user would be if the person is *female*. If the user answers "yes", the question *pregnant* would be asked. If the user answers "no", then the fact *pregnant* would become *false* and no question about it would be made. The only difference of *needs_false* is that the behaviour is the inverse one with respect to the answer to the first question. These relations can be declared between facts of any type and non-numeric facts. For a set fact, we consider its value to be *true* when all the elements of the set are *true*; we consider it to be *false* when all the elements of the set are *false*. Similarly, for linguistic facts we consider its value to be *true* when all the linguistic values are *true*; and *false* when all the linguistic values are *false*.

Needs_value: Similar to *needs_true* and *needs_false* relations but between facts of any type and numeric facts. If the numeric fact gets a value then the question of the fact is made. If the numeric fact does not get a value, the value of the fact holding the relation will be *unknown*.

Needs_quantitative: As saw in Section 3.1.1 the value of fuzzy and linguistic facts depends on a numeric fact. This relation with a quantitative fact (numeric) is then mandatory to inform the system to first ask for the value of that fact and then *fuzzify* it with the characteristic functions.

Needs_qualitative: Similar to the previous relation, it asks for the value of a qualitative fact (fuzzy or linguistic) and then *defuzzifies* that fact value to produce a number.

As a common rule for all relations r of one of the previous types, given $r(a, b)$, if b's value is unknown then a will be unknown as well.

3.2 Rules

The syntax of **Milord II** rules is given in Figure 8. The rules are composed of an identifier, the premise (that is, a conjunction of conditions), the conclusion, the certainty value of the rule, and, optionally, a documentation string. The certainty value of a rule is a linguistic term belonging to the local logic of the module.

rule	::= *ruleid* **If** premisse-rule **Then** conclusion-rule [*documentation*]
premisse-rule	::= condition-rule **and** premisse-rule | condition-rule
condition-rule	::= conditio | **no** (conditio)
conditio	::= operator (expression, ..., expression) |
	expression operator expression |
	pathpredid | *ltermid* | **true** | **false**
expression	::= operator-arit (expression, ..., expression) |
	(expression operator-arit expression) |
	number | pathpredid
operator	::= $<$ | $>$ | $<=$ | $>=$ | $==$ | $/=$ | **int** |
operator-arit	::= $+$ | $-$ | $*$ | $:$
conclusion-rule	::= **conclude** rconclusion **is** cert-value
rconclusion	::= *predid* | *predid* $==$ *symbol* | **no** (*predid*) | **no** (*predid* $==$ *symbol*)

Figure 8: Syntax of rule definition.

Facts are the basic elements used to build rules. They appear in the conditions and in the conclusion of rules. Class facts have no value and do not appear in the rules. The evaluation of a condition or a conclusion must always be an interval of truth-values. Then, in the case of facts whose type is neither boolean nor many-valued nor fuzzy, the language is provided with a set of predefined predicates that apply on them to produce as result intervals of truth values, and hence can be used as conditions of rules.

3.2.1 Conditions

First of all we explain the conditions of the rules. In order to give an understandable explanation to the way conditions of rules are written, we do not consider conditions containing paths. The conditions of a rule can be written in affirmative form (`condition`) or in negative form (`no(condition)`).

The elemental conditions can be:

1. A truth-value belonging to the set of linguistic truth values of the local logic of the module, including `true` or `false`. This is the most simple possible condition. Truth values evaluate to degenerated intervals, that is $\alpha \in A_n$ evaluates to $[\alpha, \alpha]$. A possible use of this conditions is to give initial values to facts or to limit the conjunctive value of the premise of a rule. For instance, the next rule unconditionally gives the value $[definite, definite]$ to the fact *ciprofloxacine*.

   ```
   R001 If true then conclude ciprofloxacine is definite
   ```

2. The name of a fact of type boolean, many-valued or fuzzy. For instance,

   ```
   R002 If pregnant then conclude vancomicine is possible
   ```

   where `pregnant` is a boolean concept.

3. Predicates over expressions containing facts of type numeric, set or linguistic.

   (a) *Numeric predicates*
   A numeric expression is composed by numbers, numeric facts and arithmetic operations (+,-,*,:) shown in Table 1 with the meaning expressed in the first column. The evaluation of an expression of this type returns a number over which we can predicate with the predicates shown in Table 2, where the meaning is expressed in the first column.

	numeric	**set**
`a + b`	$a + b$	$A \cup B$
`a - b`	$a - b$	no sense
`a * b`	$a \times b$	$A \cap B$
`a : b`	$a \div b$	no sense

Table 1: Arithmetic operations between expressions.

   The valid predicates are < (less), > (greater), <= (less or equal), >= (greater or equal), = (equal), and / = (different). We decided to overload these predicates to reduce the number of total predefined predicates of the language. An example of use of these numerical predicates is:

   ```
   R003 If temperature > 39  then conclude fever is definite
   ```

   (b) *Set predicates*
   We can build set expressions in a similar way as numerical expressions taking into account that the operations "+" and "*" are interpreted as the fuzzy set union and the fuzzy set intersection respectively. "-" and ":" are not applicable over set facts (see the second column of Table 1). The valid fuzzy predicates are `<` (subset), `>` (superset), `<=` (subset or equal), `>=` (superset or equal), `=` (equal), `/=` (different) and `int` (intersection degree). The binary predicates apply over the evaluations of two

	numeric	set
`a < b`	$a < b$	$A\tilde{\subset}B$
`a > b`	$a > b$	$A\tilde{\supset}B$
`a <= b`	$a \leq b$	$A\tilde{\subseteq}B$
`a >= b`	$a \geq b$	$A\tilde{\supseteq}B$
`a = b`	$a = b$	$A\tilde{=}B$
`a /= b`	$a \neq b$	$A\tilde{\neq}B$
`a int b`	no sense	$A\tilde{\cap}B \neq \emptyset$

Table 2: Operations between expressions.

expressions of the same type. The evaluation of these predicates is, as before, an interval of truth-values. We give the sense of the predicates for fuzzy sets in the second column of Table 2. To get the mathematical details of the predicates refer to [21]. For instance, considering the following rule,

```
R004 If treatment int (etambutol,ganciclovir) then conclude ...
```

the operation `int` (fuzzy intersection degree) is applied between two fuzzy sets, the first one corresponding to the value of the fact `treatment` (that is, a membership value for each element of (`etambutol, aciclovir, ganciclovir`)), and the second one a fuzzy set with the elements `etambutol` and `ganciclovir` with value *true* and the others (in this case only `aciclovir`) with value *false*.

(c) *Linguistic predicates:* Linguistic variables, together with linguistic values, are used to build predicates of the form "*linguistic_variable* `is` *linguistic_value*". For instance:

```
R004 If fever is high then conclude ...
```

Given a numerical value s for the variable $fever$, the predicate `fever is high` returns the value of $\mu_{high}^{fever}(s)$.

It is also possible to define more complex linguistic predicates of the form "*linguistic_variable* `is (` *value_1* `or` *value_2* `or` ...`)`". For instance:

```
R004 If fever is (medium or high) then conclude ...
```

Now, given a value s, the predicate `fever is (medium or high)` returns the value of: $max(\mu_{medium}^{fever}(t), \mu_{high}^{fever}(t))$.

3.2.2 Conclusions

The syntax of the conclusion of rules is simpler than conditions. Notice that in the conclusion of rules no paths can appear because the module can only conclude local facts.

Conclusions may appear on affirmative or on negative form. Concluded facts must be of many-valued or set types. e.g.:

```
R005 If ... then conclude t=y is definite
```

For set facts the value of every element is concluded independently. For instance, for the fact `t` of type `(x,y,z)` the rule `R005` give the value `definite` as the membership degree of `y` to the fact `t`.

4 Local Logics

The logical foundations of **Milord II** have been the main topic of several papers, for instance [14, 18, 3, 2, 12]. It is based on a family of many-valued logics. Each logic is determined by a particular algebra of truth-values. Any of such truth-values algebra is completely determined as soon as the set of truth-values and the conjunction operator are fixed. In **Milord II** these logics can be local to each module. To declare a local logic (see Figure 9) the user has to write down[5] the set of linguistic terms and the conjunction operation best adapted to the solution of the problem the module represents.

logcomp	::= [lingtermdef] [conjunction] [renaming]
lingtermdef	::= **Truth values** = (ltermidlist)
ltermidlist	::= *ltermid* , ltermidlist | *ltermid*
conjunction	::= **Conjunction** = truth-table
truth-table	::= **Truth table** (arrows)
arrows	::= (termlist) | arrows arrows
renaming	::= **Renaming** lrenames$^+$
lrenames	::= pathid/*ltermid* ==> cert-value
cert-value	::= *ltermid* | [*ltermid* , *ltermid*]

Figure 9: Syntax of local logic definition.

`lingtermdef` is the declaration of an ordered set of n linguistic terms $A_n = a_0 \preceq a_1 \preceq \cdots \preceq a_{n-1}$, where a_0 stands for *false* and a_{n-1} for *true*. For instance,

```
Truth values = (false, unlikely, may_be, likely, true)
```

`conjunction` is the declaration of the truth table of a conjunctive connective over the previously defined term set. A possible such operator[6], representing the *min* conjunction, could be,

```
Conjunction = truth table
             ((false false    false    false    false)
              (false unlikely unlikely unlikely unlikely)
              (false unlikely may_be   may_be   may_be)
              (false unlikely may_be   likely   likely)
              (false unlikely may_be   likely   true))
```

[5] In fact, there is a default logic for the modules that do not contain a local logic declaration (see Appendix B).

[6] This operation must hold the properties of a T-norm.

Different modules can have different local logics. We allow this because a very important part of any problem solving method is the way the programmer will deal with the uncertainty of the problem. And this may be particular to each subproblem: a richer set of linguistic terms can help in giving more precise answers to queries; different connectives represent different rule interpretations, and hence different deductive behaviours; even changing just the name of the terms from a module to another can make the knowledge represented in them more readable.

The main problem that has to be addressed in a system with local logics is how modules communicate and with are the properties that are to be held in that communication. That is, how a module has to interpret the answer to a query made to a submodule with a different logic. The topic is of great importance for the uncertainty research community [1]. The practical aspect of it is that any language providing such local logic facilities has to permit a way of defining the relation between the values of different logics. In the case of **Milord II** we do so by the declaration, in the local logic of a module, of a renaming function that maps the linguistic terms of the local logics of submodules into intervals of the linguistic terms of the module.

For instance, the translation of the terms of a module B, B_7 = {impossible, few_possible, slightly_possible, possible, quite_possible, very_possible, definite} to a module containing a local logic with terms A_5 = {false, unlikely, may_be, likely, true} could be expressed as the following sentence,

```
Renaming B/false ==> impossible
         B/unlikely ==> [impossible, few_possible]
         B/may_be ==> [few_possible, very_possible]
         B/likey ==> [very_possible, definite]
         B/true ==> definite
```

Milord II checks whether the proposed translation between modules satisfies the requirements expressed in [2, 1]. If the local logics are the same the identity renaming function is assumed by default and no renaming declaration is necessary.

5 Reification and reflection

Before explaining how the control component is defined in **Milord II** we think that it is necessary to explain which is the relation between the deductive component and the control component in a **Milord II** module. At any moment of the execution of a module, either the deductive component or the control component is active and executing. It is the evaluation strategy of the module which determines when the execution focus has to change from one component to the other. Prior to a change in the execution focus some activity is undertaken to prepare the correct execution context for that new focus. When we stop executing the deductive component and before we start the control component execution a *reification* procedure is activated. In the reverse case, after stopping the execution of the control and before starting the execution of the deductive component a *reflective* procedure is executed.

These procedures modify the working memory (theory) of the component that is going to start by adding formulas to it. In this section we explain which for-

mulas are added and under which conditions. Programmers cannot modify the way the reification and reflection procedures work, as it is fixed in the **Milord II** architecture.

5.1 Reification

When the deductive component stops we have to update the control component working memory so it has an up-to-date representation of the state of the deductive component. The reification procedure adds meta-predicate instances. These instances will cause that the adequate meta-rules activate. In Figure 10 you can see which is the syntax of general meta-predicates in **Milord II**.

gexpr	::= *predid* (term , ... , term)
term	::= $*varid* | *symbol* | *termid*(term, ..., term)

Figure 10: Syntax of general meta-predicates definition.

The part of the module state that does not change during execution is reified only the first time the reification procedure is called. We call this reification part *static*. The *dynamic* reification corresponds to that part of the state that changes during execution. *Static* and *dynamic* reification generate instances of predefined meta-predicates (see Figure 11).

5.1.1 Static

The components of the module state that are persistent during the execution are: the programmer defined relations between facts, the threshold and the rules. All these components are initially reified. Although the submodules of a module are not persistent, the initial submodules are also reified.

Relations: The name of the relation used in the definition of facts, user-defined or system-defined, becomes a binary meta-predicate identifier. The two arguments correspond to the name, with the appropriate path prefix, of the facts being related. For instance the relation `relationid` between *fact1* and *fact2* becomes the next meta-predicate instance:

```
relationid(fact1, fact2)
```

Threshold: It asserts a meta-predicate instance to represent the threshold of the current module and those of its submodules.

```
threshold(lingterm)
threshold(submodule,lingterm)
```

Rules: There is a kind of evaluation strategy named *reified* (see Section 6.1) that needs to know the rules of the deductive component at the meta-level. In modules with that strategy the rules are then also reified. For instance consider the following rules concluding a many-valued fact `c` and a set fact `d`:

mexpr	::= known | mrel | msubmod | mthres | card | atom | member | eqdif | moper | int | setof | pos | gexpr
symorvar	::= *symbol* | **$***symbol*
vpath	::= symorvar | symorvar/vpath
known	::= **K** (fact, interval)
fact	::= factex | **not**(factex) | **implies**(list,list)
factex	::= vpath | **=**(vpath, symorvar)
interval	::= **$***symbol* | **int**(symorvar, symorvar)
mrel	::= *relationid*(symorvar, vpath)
msubmod	::= **submodule**(symorvar, symorvar) | **submodule**(symorvar)
mthres	::= **threshold**(symorvar, symorvar) | **threshold**(symorvar)
card	::= **cardinal**(list, symorvar)
list	::= **$***listid* | (listelem)
listelem	::= *elemid* | *elemid* , listelem
atom	::= **atom**(list)
member	::= **member** (symorvar, list)
eqdif	::= **equal**(listorsym,listorsym) | **diff**(listorsym,listorsym)
listorsym	::= symbol | list
moper	::= loper(symorvar,symorvar)
loper	::= **lt** | **le** | **eq** | **neq** | **ge** | **gt**
int	::= **intersection**(list,list)
setof	::= **set_of_instances** (**$***var*,term,**$***var*)
pos	::= **position**(symorvar,list,symorvar)

Figure 11: Syntax of meta-expression definition.

```
If a and b then conclude c is s
If a and b then conclude d=x is qp
```

These rules are reified in the following form:

```
K(implies(prem(A,B), C), int(S,S))
K(implies(prem(A,B), =(D,X)), int(QP,S))
```

Notice that the last list represents and upper interval of truth-values (considering the truth-value *true* is the linguistic term `S`).

Submodules: Informs the meta-level of all the submodules of the current module by means of instances of a meta-predicate called *submodule* (meta-predicate with only one argument), and the submodules of every submodule of the current module (the same meta-predicate name but with two arguments).

```
submodule(submodule)
submodule(submodule,subsubmodule)
```

5.1.2 Dynamic

Every time a reification is performed the current values of facts are asserted as instances of the meta-predicate K as follows:

Many-valued, Fuzzy and Boolean: The value of a fact, `fact`, of any of these types is an interval of truth-values, the instance of K being reified is the following (the second interval is the negation of the first one):

```
K(fact, int(lingterm1, lingterm2))
K(not(fact), int(lingterm3, lingterm4))
```

Set and Linguistic: Given a fact, `fact`, we reify two instances (again an affirmative and a negative one) for every element or linguistic value.

```
K(=(fact, value_i), int(lingterm_1, lingterm_2))
K(not(=(fact, value_i)), int(lingterm_1', lingterm_2')) ...
```

Numeric: For numeric facts we reify the next instance of K meta-predicate:

```
K(=(fact, number), int(true, true))
```

In some cases the control can inhibit (filter) submodules or declare new submodules of a module (see Section 6.5), hence the next two meta-predicates:

Submodule: With the same syntax of static reification, it informs the meta-level of the new submodules of the current module.

Filtered: It informs the meta-level of the the submodules that are filtered by meta-rules.

```
filtered(submodule)
```

5.1.3 Reflection

When we change the execution focus from the control component of a module to its deductive component the reflection procedure is in charge of informing the deductive component about the modifications in the theory and modular structure generated during the execution of the control component. Hence, for each meta-predicate instance of particular meta-predicates and for each action scheduled a procedure is undertaken to change adequately the deductive component theory.

K(f,*int*) Fact f's value is updated to become *int*. It is a destructive operation and no truth maintenance mechanism is put into action. It is, in general, a dangerous operation. Programmers should avoid it.

Inhibit rules [*relationid*] *pathid* The rules containing the fact *pathid* in their premises, or optionally any fact related by *relationid* to *pathid* are eliminated from the theory.

Prune *pathpredid* All rules having a fact in their premise directly or indirectly related with the fact *pathpredid* are eliminated from the theory.

Filter $amodid^+$ All submodules contained in the expression $amodid^+$ are eliminated from the structure of the module. Again, this is a destructive operation, no truth maintenance mechanism is engaged to retract those deductions based on facts exported by the eliminated submodules, which from this moment on will be hidden to the module.

Order $amodid^+$ **with certainty** *cert_value* This action is only effective in eager modules. The order in the execution of submodules will be the result of weighting the certainty value of different *order* actions, the relative position of modules in those actions, and the writing order.

Open(amodid), {Module, Inherit}(*term***)** The necessary linking procedure is engaged to generate the submodule (that can be in general a complex activity) and to update the module structures representing the hierarchy, so that from then on the so generated new submodule will have the same status as if it had been defined statically.

6 Control knowledge

The control knowledge is the declarative component of a module responsible for several activities: meta-reasoning, order in evaluation of facts and rules, determination of which rules to use and other control tasks. The control knowledge component is composed (see Figure 12) by the local declaration of the type of evaluation, the threshold and a set of meta-rules that controls the deductive (rules) and the structural (hierarchy) components of a module.

control	::=	**Control knowledge**
		[**Evaluation type:** evaltype]
		[**Truth threshold:** *ltermid*]
		[deduccnt] [structcnt]
		end control
evaltype	::=	**lazy** | **eager** | **reified**

Figure 12: Syntax of control definition.

6.1 Evaluation Strategy

A module execution computes answers to queries being made to it. A query consists of asking for the value of an exportable fact. The way the module executes to obtain that answer can be different, from the point of view of the queries the module will do in turn to the user, and the way the submodules will be queried as well.

The three possible evaluation strategies in **Milord II** are called lazy, eager and reified strategies. Before explaining in detail these strategies in Section 8 we give here an intuitive idea of their main differences:

Lazy: A module with this evaluation strategy asks the user for values for the facts in the import interface, and queries for values of facts in the submodules' interface only if necessary. This strategy is used by default.

Eager: Given a query to an eager module the following actions are done: it asks the user for the value of all the imported facts of the module, and it also asks for the values of all the exported facts of its submodules.

Reified: This kind of evaluation strategy asks questions in the same form as the eager evaluation strategy, but no actions are made at deductive level. The rules of the deductive component of a module are reified. It is then the control which gives sense to the reified rules by simulating inference rules by means of meta-rules.

6.2 Threshold

Milord II allows the programmer to give a value to a parameter that controls the minimum truth-value that a premise has to evaluate to in order to fire a rule in that particular module.

This parameter is named the *threshold* of a module. Its value is a linguistic term and the default value is the second term a_2 of the chain A_n of truth-values (see Section 4). MYCIN [23] had certainty factors lying in the interval $[-1, 1]$, and used a similar approach by having a threshold to fire rules of 0.2. An example of threshold declaration is:

```
Truth Threshold: may_be
```

As another consequence of the threshold parameter all facts with a lower truth-value less or equal than the threshold become *unknown*.

6.3 Meta-rules

The last component of the control knowledge consists of two sets of meta-rules, that are responsible for tuning the inference at the object-level, or of implementing it, in the case of a reified evaluation strategy, and of managing the hierarchical structure of submodules by adding or cancelling dependencies. Meta-rules in both sets have a similar syntactical structure, that is, a conjunction of meta-predicates as premise and a set of meta-predicates as conclusion. The only difference is on the meta-predicates allowed as conclusion in each set. In this section we describe the behaviour of the predefined meta-predicates used in the premises and in the conclusions. Apart from them, any user-defined meta-predicate can be used in the premises or in the conclusions. Negation in the conditions is defined as negation by failure.

6.3.1 Meta-predicates in premises

The remaining pre-defined meta-predicates, not yet presented in Section 5, of **Milord II** with their meaning is presented next.

cardinal(list,n): It evaluates to true if the length of *list* is equal to *n*. The first argument must be grounded before the evaluation is performed. If the second is a free variable, this predicate instantiates it. For instance `cardinal((a,b,c),3)`evaluates to true.

atom(expression): It is a predicate that returns the value true if the *expression* evaluates to an atom. *expression* must be grounded.

member(el, list): Evaluates to true if the element *el* belongs to *list*. For instance, `member(d,(a,b,c))` returns false. Both arguments must be grounded.

equal(exp1,exp2), diff(exp1,exp2): These operations compare two expressions returning true if they are equal or different respectively. Both expressions must be grounded.

{lt, le, eq, neq, ge, gt}(exp1,exp2): These operations (less than, less or equal, equal, not equal, greater or equal and greater than, respectively) allow us to compare numbers or linguistic terms. For instance `le(possible,definite)` returns true if *possible* is less or equal than *definite* in the ordering established in the local logic of the module. Likewise `ge(4,5)` will evaluate to false. Again the expressions must be grounded.

intersection(list1,list2): Returns true if the intersection of the two argument lists is not empty.

set_of_instances(var1, expression, var2): Given an *expression* containing the variable *var*1, the variable *var*2 will be bound to a list containing all the instances of *var*1 that make the *expression* true. The expression can contain any meta-predicate expression combined by the next reification names for the connectives: *conj* for $\wedge$ and *neg* for $\neg$. *var*1 and *var*2 must be free. For instance, given the following predicate

```
set_of_instances($x,Conj(K(=(f,$x),int(s,s)),
                         member($x,list(a, b, c))),
                $values)
```

the variable *values* will contain all the elements of the set fact *f* that are true and are either a, or b or c..

position(list, int, exp): This meta-predicate is true if the value in list *list* at position *int* is the same as the value of *exp*. If *exp* is a free variable , the result will be true and *exp* will be bound to the value of *list* at position *int*. If the value of *int* is out of the limits of *list* the predicate will return the value false.

deduccnt	::=	**Deductive control:** mrr$^+$
mrr	::=	metaid **If** premisse-meta **Then** filter-mrr$^+$
premisse-meta	::=	mexpr **and** premisse-meta | mexpr
filter-mrr	::=	**inhibit rules** [relation-id] pathpredid |
		prune pathpredid | **conclude** gexpr | **conclude** known

Figure 13: Syntax of deductive control definition.

6.4 Meta-rules: Deductive Control

The deductive control (see Figure 13) may affect the deductive knowledge of a module by inhibiting rules or deducing instances of the meta-predicate K. Moreover, instances of any user-defined meta-predicate can be generated. These meta-rules are responsible for the implementation of a type of meta-reasoning based on changing the object level theory in two ways: by reducing it (eliminating some rules), or by changing it (deducing instances of the meta-predicate K). Next you have a description of the meta-predicates allowed in the conclusions of this type of meta-rules. The action to be taken after a meta-rule deduces an instance of these meta-predicates is explained in Section 5.1.3.

Inhibit Rules: This action inhibits all the rules containing the fact *pathpredid* in their premises. Optionally we can introduce a name of relation *relation-id*. Then the rules inhibited will then be those containing in its premises a fact related with *pathpredid*.

Prune: It inhibits all the rules belonging to the deductive tree of the fact *pathpredid*.

Conclude: Here users are allowed to conclude any meta-predicate. Only when the meta-predicate is the predefined K an action consisting on overwriting a fact value at the object level will be undertaken in the reflection phase. The remaining instances of user-defined predicates will remain as part of the meta-level state, and will eventually be used as conditions of other meta-rules to be fired in this and in posterior activations of the meta-level component.

6.5 Meta-rules: Structural Control

The meta-rules of the structural control (see the Figure 14) are designed to modify the hierarchy of a module by inhibiting modules or by declaring new ones (dynamic modules); they can also stop completely the execution of the program.

Filter: A meta-rule can inhibit (filter) submodules of a module. That means that all the facts exported by the filtered submodule will be from then on considered to have the value *unknown*.

Order: When we use eager evaluation in a module, the order of evaluating the submodules is by following their writing order. This predicate permits to

structcnt	::=	**Structural control:** mre$^+$
mre	::=	metaid **If** premisse-meta **Then** filter-mre
filter-mre	::=	**filter** amodid$^+$ |
		order amodid$^+$ **with certainty** cert-value |
		Open (term) | **Module** (term) | **Inherit** (*amodid*)
mrx	::=	metaid **If** premisse-meta **Then** exception
exception	::=	**definitive solution** predid |
		stop

Figure 14: Syntax of structural control definition.

change this order when a set of conditions hold. The truth degree is used in the combination of different instances of the meta-predicate. The real order will be a combination of partial orders weighted by the truth degree.

Open, Module and Inherit These declarations are equivalent to the corresponding normal submodule declarations, but they are performed dynamically. The following example (from *Spong–IA*) shows the dynamic creation of a submodule by means of the dynamic instantiation of a generic module. Given a value $\$z$ for the set fact $DM/taxon$ with certainty value interval $[\$min, \$max]$, and with $\$min$ greater than the threshold value of the module DM,that is, greater than $\$cut$, and if $\$z$ is a submodule of DM, then a new submodule is created, with local name $\$z$ and with body the instantiation of the generic module $Refinement\_method$ with the modules $DM/\$z$ and T as arguments.

```
M0001 if K(=(DM/taxon ,$z), int($min,$max)) and
         threshold(DM, $cut) and gt($min,$cut) and
         submodule(DM, $z) then
                  Module(=($z,Refinement_method(DM/$z, T)))
```

Definitive Solution and Stop: These are exceptional actions. In some cases the programmer wants to stop the execution by assigning a value to a fact (definitive solution) or aborting (stop) when an unrecoverable situation is reached, e.g. the problem being solved is out of the scope of the system.

7 Modular Language

As already mentioned in Section 2 modules are the structural unit of **Milord II**. A Knowledge Base consists of a set of modules, with each module containing submodules, and submodules containing subsubmodules, ... defining a hierarchy. There are two types of modules: plain modules, as explained in Section 2, and generic modules, that are to be considered as functions between plain modules. That is, given one, or more, modules as argument a generic module gives a plain module as result. To complete the picture of the modular language component a set of operations between modules is provided.

7.1 Generic Modules

When a plain module contains a set of submodules, one way of looking at what the module performs is by seeing it as a combination of the results of the different submodules. The definition of generic modules opens to the user the possibility of defining specific, and reusable, operations of composition of modules. A generic module is indeed an abstraction of a combination by means of giving names to the submodules that will be obtained only at application time. Generic modules are then operations (or functions) on modules. The technique to define generic modules is the same as to define functions, that is, it consists of the isolation of a piece of program, or module, from its context and its abstraction by specifying:

1. Those modules upon which the abstracted module may depend (requirements or parameters of the generic module).

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

The obvious referent of this technique is functional programming [16, 4], where such abstractions (functions) form the basic program units. The functional body defines how to compute the output (results) in terms of the input (requirements). In Figure 15 we extend the syntax of module declarations of Figure 1 adding the generic modules declarations (the syntax is still not complete).

PROGRAM	::= moddecl$^+$
moddecl	::= **Module** *amodid* [([paramlist])] [≡ bodyexpr]
bodyexpr	::= **begin** decl **end** | pathid [([iparamlist])]
paramlist	::= *amodid* ; paramlist | *amodid*
iparamlist	::= bodyexpr ; iparamlist | bodyexpr

Figure 15: Syntax of generic module definition.

A method for building large KB systems consists of applying generic modules to previously built plain modules. Keeping the common parts in a generic module we can save code and time, and make the code much more understandable. The instantiation of a generic module over a set of arguments generates a plain module, and hence it can appear in the code in the same places as a module declaration does.

Consider the following example of microbiological analysis of samples for pneumonia diagnosis (*Bacter–IA* application). Some data can be obtained from a gram analysis of a sample (for instance, `DCGP`). These data can be obtained from different gram analysis over different samples (of sputum, of lung, or bronchoaspirated) to deduce the germ causing the illness (for instance `pneumococcus`). In this case it is not necessary to define a different problem solution for each type of analysis; it would be enough to define a generic problem solution depending on the kind of analysis.

```
Module Find_Germ (X) =
  Begin
  Export pneumococcus
  Deductive knowledge
    Dictionary: ...
    Rules: R001 If X/DCGP then conclude pneumococcus is possible
           ...
  End deductive
End
```

The parameters of a generic module are, as we said before, abstracted submodules. These parameter names are unbound until the instantiation of the generic module. If we want to refer to the exported facts of the submodules that will be bound at application time we must build a path using the names of the parameters (for instance, in the rule of module `Find_Germ` a reference to the fact `DCGP` of a module eventually bounded is written `X/DCGP`). An example of the instantiation of the generic module seen above with data from a sputum sample is the following:

```
Module Find_Germ_Sputum = Find_Germ (Sputum_Gram)
```

The parameters of a generic module are submodules hidden outside the generic module, where hidden has the standard meaning and no access is allowed to the export interface or submodules of a hidden submodule. For instance, the submodule `Sputum_Gram` is hidden outside the new module `Find_Germ_Sputum`. So a reference like `Find_Germ_Sputum/Sputum_Gram/DCGP` will be detected by the compiler and an error will be raised.

A generic module makes use of the exported facts of the module bound to its module parameters, so a parameter cannot be bound to any module, but only to modules exporting the facts required in the body of the generic module. For instance, the module `Sputum_Gram` must export the facts needed in the rules of the generic module `Find_Germ`.

Milord II supports the process of incremental KB building by means of generic modules. Hence, whenever the definition of a generic module changes, the changes are 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 is automatised by the compiler[5], so that the user gets rid of this task.

7.2 Operations between Modules

Top-down programming methodology is sometimes related to an incremental specification of problems. We are interested in including in the language a set of operations to assist programmers in the process of development, beginning with the first prototype and ending with the final version of the program.

Incremental programming consists in writing a first (rough) prototype, test it, then write a second as a modification (refinement) of the first, and go on until a final version is achieved. In classical programming languages the way of doing it is by changing the code of the program whenever a non-desired behaviour is observed ora more specialised performance is required. But, the complete code has

to be ready and at hand before performing any testing. A step forward in helping programmers in **Milord II** is by requiring: 1) that the partial specifications of modules, incompletely defined modules, must be executable to test them, and 2) a set of operations between modules for overseeing this process of incremental building of programs.

If you observe the syntax of modules you can notice that many components of modules are optional. Any module declaration is executable. For instance, we can declare modules without control component, or without import interface, etc. In some cases we do not declare some components of modules because we want to define them incrementally. For instance, if we execute a module which contains only an export interface, it will answer to all the questions about the value of exported facts with the constant *unknown*. Later on we can fill in the module with code details by *refining* it.

When we program a new version of a previous program we are interested in checking and declaring what is the relation with the old version. In **Milord II** this relation is defined between modules and can be either a refinement, a contraction or an expansion. Roughly speaking, we say that a module is a refinement of another one when the set of accessible[7] facts is the same that the previous one but the code is a particularisation. When we expand a module the accessible facts in the next version will be extended, and reduced when we constrain it.

In Figure 16 we extend the previous syntactical declarations of modules in Figure 15 with these new module operations. The symbols ":", ">" and "<" stand respectively for the module refinement, expansion and contraction operations. They can be used in all module declarations including the parameter declaration of generic modules.

PROGRAM	::= moddecl$^+$
moddecl	::= **Module** *amodid* [([paramlist])] [modoper modexpr] [$\equiv$ modexpr]
modexpr	::= bodyexpr modoper modexpr | bodyexpr
paramlist	::= paramlist ; paramlist | *amodid* modoper modexpr
iparamlist	::= modexpr ; iparamlist | modexpr
modoper	::= : | $\geq$ | $\leq$

Figure 16: Syntax of operations among modules definition.

All these modular operations are based in three basic functions: enrichment verification, inheritance and information hiding. Next, we explain the refinement of modules that is the most important operation and makes use of these three basic functions in its definition. The other two operations are slight modifications of this refinement operation.

[7] Remember that the accessible facts of a module are the facts belonging to its export interface and those of the export interfaces of its submodules.

7.2.1 Refinement

Modules are the computational counterpart of the abstract units, usually called tasks, in which a programmer decompose a complex problem solving task [9]. This abstract units are characterised basically by the goal (query) they have to achieve. Hence, when designing a module the first decision is the goal, or the set of goals, that module will solve. In **Milord II** these goals are represented by the set of accessible facts of that module. Then, when designing a particularization of the module, that is, when filling it with more contents, we must keep the same set of goals, as far as the new module is still an implementation of the task that is being solved. The refinement operation guarantees that the new generated module fullfills this [15]. Consider the following example that completes the example in the previous Section:

```
Module Sample = Begin Export  DCGP, CGPC, CGPR, GNG, CBNG End

Module Sputum_Gram : Sample =
  Begin
       Import Sputum_class, Sputum_Gram
       Export DCGP, CGPC, CGPR, GNG, CBNG
       Deductive knowledge
            Dictionary: ...
            Rules: R001 If Sputum_Gram = (DCGP_MC)
                        then conclude DCGP is definite
                   ...
       End deductive
  End
```

The module `Sample` only contains an export interface. The second expression declares that the module `Sputum_Gram` is a refinement of the module `Sample`.

This is the idea of incremental programming, all the modules that are refinements of the module `Sample` have the same export interface with, eventually, differences in other components that allows the module to obtain better, or different, results for the exported facts than the module `Sample`.

The refinement operation is specially useful when we declare generic modules. Remember that the instantiation of a generic module implies binding parameter names to submodules. The resultant module should use the exported facts of the submodules bound. It is obvious that not all the modules can be used to instantiate a generic module, because the code of the generic module will depend upon particular exported facts of those submodules. For instance, we could modify the previous declaration of the generic module `Find_Germ` as follows:

```
Module Find_Germ (X : Sample) = Begin
    ;;the same declarations that in the definition above
                            End
```

This kind of declaration assure us that the modules used to instantiate the generic module `Find_Germ` are only those which are refinements of the module `Sample`, that is, that have exactly its same export interface (in particular, we can assure that any argument module will export the fact `DCGP`). So, usually, any generic module will have its parameters being a refinement of a very simple module

containing just an interface. More complex requirements on the arguments can be imposed by filling with contents the module from which they have to be a refinement.

A refinement operation is the result of the composition of three operations: enrichment verification, inheritance and information hiding. We will explain these components keeping in mind the following equivalent syntactic declarations:
`Module M = M_1 : M_2` $\equiv$ `Module M : M_2 = M_1`

Enrichment Verification We say that the module `M_1` is an *enrichment* of the module `M_2`, if and only if:

1. The export interface of `M_2` is a subset of the export interface of `M_1`.
2. The submodule names of `M_2` are a subset of the submodule names in `M_1`. If a submodule name is bound both in `M_2` and `M_1`, the modules to which it is bound, let's say `M_21` `M_11`, must satisfy that either `M_11` is an empty body, or otherwise `M_11` must be an enrichment of `M_21`.
3. The local logic declaration must be the same; or empty in `M_2`.

That means that the module `M_1` can extend the export interface and the submodules of `M_2`. When a submodule is declared in both modules `M_1` and `M_2`, they must preserve the enrichment relation.

Inheritance When we declare a module as a refinement of another we usually want to maintain several components of the module being refined. To avoid the programmer to write the components to be preserved twice, an inheritance mechanism is provided in **Milord II**. The components of a module that can be inherited are: submodules, fact definitions and local logics.

When we define a refined version of a module, if we omit the body of a submodule, it will be inherited. Hence, the module `M` will inherit the bodies of the submodules of `M_2` that not are present in the declaration of `M_1`. The inheritance operation makes a copy of the non redefined elements of the dictionaries. In the case of the local logic, the module inherits the logic of module `M_2`.

Information Hiding One of the conditions that the modules have to fulfill to satisfy a refinement relation is that the accessible facts of both modules must be the same. So, after checking the enrichment of information, **Milord II** hides all new accessible facts and new submodules of the refined module (if any).

In a refinement operation, information hiding affects the export interface and the modular structure of the module created by the refinement. All the exported facts of `M_1` not present in the export interface of `M_2` are hidden in the resulting module `M`. Similarly all the submodules of `M_1` not visible in the hierarchy of `M_2` are hidden in the resulting module `M`.

Refinement is then defined as the combined action of these three operations. The other two relations between modules are slight variations of refinement.

7.2.2 Expansion and Contraction

Expansion allows to build modules as an extension of a previous version. We can extend the set of accessible facts or add submodules to the previous version. As in the refinement case to expand modules we test that the new module is an enrichment of the previous one. Inheritance of components is performed as in the refinement operation, but information hiding is not applied.

In the next example, we are forcing the argument of a generic module `Generic` to have *at least* the export predicates of `Type` in their export interface.

```
Module Generic (X > Type) = Begin ... End
```

So, the generic module will be applicable over a wider range of modules than if its argument was defined as a refinement of `Type`.

Contraction only test an inverse enrichment verification. Given the declaration `Module M = M_1 < M_2` we only test that `M_2:M_1` holds. It is not necessary to apply information hiding because the module `M_1` exports less facts than `M_2`. Inheritance of components is performed as in the refinement operation.

7.3 Special declarations

We shall complete this Section by briefly commenting on the different types of module declarations allowed in **Milord II**. *Open* and *Inherit* submodule declarations are only programming facilities and they do not belong to the primitives of the modular language. `Open bodyexpr` means that we will not give a name to the submodule, the practical consequence is that exported facts of open modules can be directly referenced, without any path prefixing them, as if they were textually defined in the module containing the open declaration, the same happens to submodules of an open submodule that are considered as if they were direct submodules of the module containing the open declararation. `Inherit A` is an equivalent expression for `Module A = A`. The complete syntax of the hierarchy component of modules is given in Figure 17.

hierarchy	::= moddecl | **Inherit** *modid* | **Open** bodyexpr | hierarchy hierarchy

Figure 17: Syntax of submodule definition.

It is easy to see that name clashes can occur when we use declarations by means of *open*. If a module has more than one opened submodule, the names of the exported facts must be different[8].

8 Operational Semantics

In this Section we describe the behaviour of modules when computing the answer to a query. As we have already noticed, the behaviour depends on the value of the

[8] **Milord II** Compiler [5] detects all these conflicts.

`evaluation type` parameter of each module. During the process of evaluating the answer to be given to a query, a module execution is engaged in several activities: make questions to users; make queries to submodules, initiating, then, similar evaluation processes; specialise rules to deduce new facts [20]; deduce new facts which are reified to the control component, ... (see Figure 18). Determining which activity is to take place next is the main task of the control associated to a module, and it is what we mean by Operational Semantics here.

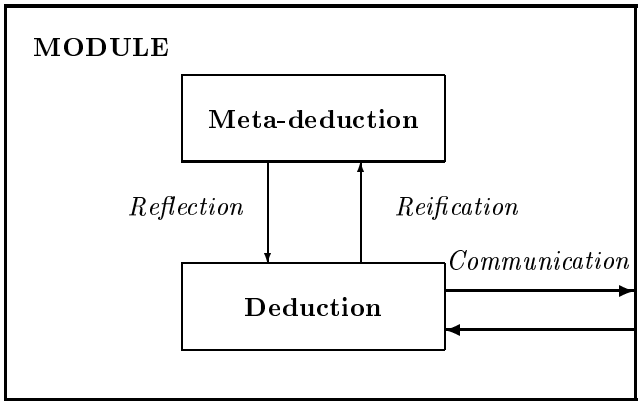

Figure 18: Some activities in the computation of a query answer.

In **Milord II** there are three types of evaluation strategies: Lazy, Eager and Reified. We explain the behaviour of the module execution for each one of these strategies.

8.1 Lazy

A module with lazy search strategy uses the fastest way to obtain the solution for a query. By fastest we understand the minimum number of new queries generated, for both the user and the submodules, to solve the current query. The strategy is based on the combined action of 1) a search procedure that, looking at the current state of the module (values of the facts and current rules, relations and functions), finds the next query that is relevant to the current query, and 2) the updating of the module state by means of the specialisation of the knowledge base. This cycle is repeated until the answer for the query is generated.

We should remember which are the components of a module that participate in giving a value to a fact. A fact receives a value from the user (if the fact appears in the list of imported facts), from a submodule (if the fact is a prefixed one), by means of *needs* relations (in the sense explained in Section 3.1.4), from a function associated to a fact (the value of the fact will be the result of the evaluation of the function), from the rules (the value will be obtained as a combination of the

conclusions of rules), or from the meta-rules (the value is the result of the reflection of a K meta-predicate instance.

8.1.1 Query

We start the evaluation process by putting a query to a concrete module of an application. This process will end up by answering with the value obtained for that query. An example of query to the module *sponges* can be $Query(sponges, taxon)$.

$$Query : Module \times Query \rightarrow Query \times Value$$

Assume that the query is q. There are two possible cases to consider:

1. $q = sm/q_{sm}$ *is a path:* If the fact is a path to a submodule sm of the current module, and if that submodule is visible (see Section 7), it makes a query to that submodule[9] $Query(sm, q_{sm})$, otherwise it raises an error.

2. q *is a fact:* If q belongs to the export interface of m it starts $Eval(q)$ in that module returning the value of fact q; otherwise it raises an error.

8.1.2 Evaluation

Eval is a procedure that finds the value of a fact f by using all the resources of a module: the user, its submodules, rules, meta-rules and so on.

$$Eval : Fact \rightarrow Fact \times Value$$

It consists in searching (*Search*) which is the information from outside the module needed to solve f, obtaining that information and finally updating (*Update*) the state of the module. This process stops when a value for f is found.

$$Eval(f) = \texttt{loop } \{Update(Search(f)); \texttt{ if } value(f) \texttt{ return } (f, value(f))\}$$

8.1.3 Search

Hence, given a query to a module, the algorithm sketched below shows how the module searches for the next question to be made to the user (*Ask* procedure) or to a submodule (*Query* procedure) that is needed to answer the current query. This algorithm returns only a fact belonging to the import interface of the current module, or a fact belonging to a submodule with its value.

This is a recursive algorithm because the initial goal produces new internal subgoals that in their turn use the same algorithm. Consider that f is the fact constituting the current query.

$$Search : Fact \rightarrow Fact \times Value$$

[9] Obviously this process can go recursively putting queries into the hierarchy of a module.

1. $f = sm/f_{sm}$ *is a path:* If the fact is a path to a submodule sm of the current module, and if that submodule is visible (see Section 7), it puts a query $Query(sm, f_{sm})$ to the submodule and returns the value so obtained; otherwise it returns an error.

2. f *has unsolved* needs *relations:* The facts used in the body of the functional expression attached to f are considered as if f had a *needs* relation with them. Therefore, the *needs* relations of a fact are those explicitly declared by the expert plus those implicitly added by the function attribute. A *needs* relation is considered to be solved when the fact with which f has the relation got a value. It recursively applies $Search$ to the first fact with which f has a no solved *needs* relation.

3. f *belongs to the import interface and has no value:* In this case it makes a question to the user $Ask(f)$.

4. f *can be deduced by means of rules:*

   (a) *Rule ordering:* Before starting the rule search we order the set of rules that are able to deduce fact f with the following criteria:

      i. Rules more specific first.
      ii. In case of rules equally specific, first those whose certainty value is higher.
      iii. In case of equal certainty degree, we use the writing order.

   (b) *Rule search:* We start a *depth first* search on the rules[10] by:

      i. Selecting the first rule of the ordered set obtained above.
      ii. Selecting the first fact in the writing order of the conditions of that rule (left to right). Then recursively apply $Search$ to that fact.

5. *otherwise:* f is *unknown.*

Notice that the algorithm $Search$ finally puts a question to the user $Ask(f)$, makes a query to a submodule $Query(sm, f_{sm})$ and returns the value obtained or, if everything fails it returns the value *unknown.*

8.1.4 Update

When the $Search$ algorithm finally returns the result of the evaluation of a fact f the system updates the state of the module with respect to that fact by means of the deduction of facts, the specialisation of rules, and so on.

Given a state of a module, and a set of facts and their values, *Update* is a procedure that returns a new state and a set of facts and values, obtained, for instance, as the result of solving a *needs_true* relation or firing a rule. This function is applied recursively on its results until there is no state change.

[10] Taking into account that the *need* relations will modify the pure depth search strategy.

$$Update : State \times Fact \times Value \rightarrow State \times (Fact \times Value)^*$$

The tasks undertaken by this procedure are:

1. *Solve need relations:* As it has been seen in Section 3.1.4 a fact with this kind of relations with another fact can be given a value depending on the value of the related fact.

2. *Solve functions:* When all the facts involved in a function attribute have value, the fact containing that function can get a value by the application of that function.

3. *Specialise rules:* New facts evaluated that belongs to the premises of rules allows the *Specialisation* of them (see next subsection), obtaining a new rule set and eventually new fact values.

4. *Reification-reflection:* Each time a new fact receives a value a reification-reflection step is undertaken. As a result of it new facts may receive values by the reflection of meta-predicate K instances.

8.1.5 Specialisation

In rule based systems, deduction is mainly based on the modus ponens: $A, A \rightarrow B \vdash B$. Modus ponens is only applicable when every condition of the premise of the rule to be fired is satisfied, otherwise nothing can be inferred. **Milord II** uses partial evaluation to extract the maximum information even from incomplete knowledge about the truth-value of the premises of a rule.

We base the partial evaluation of rules on the well known logical equivalence $(A \wedge B) \rightarrow C \equiv A \rightarrow (B \rightarrow C)$ which leads to the following boolean specialisation inference rule: $A, A \wedge B \rightarrow C \vdash B \rightarrow C$. The rule $B \rightarrow C$ is called the *specialisation* of the rule $A \wedge B \rightarrow C$ with respect to the fact A.

It is easy to see that we can specialise rules deleting the known facts from the premise. Unknown facts remain as part of the premise of the rules. For instance, suppose that we have the following rule: $A \wedge B \wedge C \rightarrow D$. Imagine that we only know A and C are true. Then the specialised rule is: $B \rightarrow D$.

Using modus ponens inference rule, the answer to the goal D would have been *unknown* because we do not know if the fact B is true or false. If we give the above rule as an answer, then its interpretation is the following: The truth-value of D depends on the truth-value of B, if B is *unknown* so will D be (open world assumption).

This boolean case may be considered of little interest, but we can extend this specialisation concept to the more interesting uncertainty calculus. Let's introduce the definition of what we call Specialisation Inference Rule (SIR). Given a fact A with certainty value α, and a rule with certainty value ρ, then

$$(A, \alpha), (A \wedge B \rightarrow C, \rho) \vdash (B \rightarrow C, \rho')$$

where $\rho' = conjunction(\alpha, \rho)$ is the new truth value of the specialised rule[11]. Conjunction is the connective declared in the local logics of a module as seen in Section 4.

Now consider the following example of rule: `If a and b then conclude d is likely`. Imagine we know that `b` has the value `may_be`. The resultant specialised rule will be: `If a then conclude d is may_be` after applying the table for *conjunction* of Section 4. Notice that the truth-value of the rule has changed because of the uncertain value of the fact `b`. This is important when the KB contains a set of rules also deducing `d`. The specialised rule will change its place in the priority order affecting the search strategy presented above.

$$Specialization : Rule^* \times (Fact \times Value)^* \rightarrow Rule^* \times (Fact \times Value)^*$$

The specialisation of the whole deductive knowledge of a module consists on the exhaustive specialisation of its rules. Rules whose conditions contain facts with known values are replaced by their specialisations, in particular, rules that only have one known condition will be eliminated and their conclusions added as a known fact in the module. This new fact will be used again to specialise the knowledge base. The process will finish when the deductive knowledge has obtained an answer for the query or there are no more possibilities for deducing the fact by means of rules.

8.2 Eager

The eager strategy is radically different. Now we do not have any economical criteria to find the questions. Given a query to a module, that module asks all the facts of its import interface, and all the facts belonging to the export interfaces of its submodules.

The ordering of these goals is that of the declarations given by the expert, that is, first the facts of the import interface of the module following their writing order, then the facts of the export interface of its submodules. Modules are executed following the writing order unless ordering meta-rules changing it have been fired. In this kind of search only the *needs* relations are taken into account, in the same way as before, when asking questions to the user.

The difference with *lazy* evaluation is the *Search* procedure. Only after obtaining the values for all the facts (from the user or from the submodules of the current module), the *Update* procedure is performed.

8.3 Reified

It is the most flexible evaluation strategy. Rules are reified into the meta-level component as meta-predicates of type `rule` (see Section 5.1.1). If the query is a path, contains a function or appears in the import interface, then the behaviour is

[11] This is a simplified view because, as the reader already knows, **Milord II** deduction is based on intervals of truth-values.

as in the eager case. Otherwise, all the deduction is made at the meta-level, and the value of the query will be the result of the reflection of a `K` meta-predicate.

All the procedures of *eager* evaluation are performed except that of point 3 of the *Update* procedure (deduction by means of rules).

9 Conclusions and discussion

In this paper a complete description of the language syntax of **Milord II** has been undertaken. In this sense this paper complements others, already referenced throughout the paper, that concentrate on much more formal aspects of the language. **Milord II** has proved to be a very expressive language to solve problems in very different domains. So, in the near future, we plan to extent its capabilities to distributed environments by wrapping **Milord II** modules inside computational agents.

Milord II, and its previous language version *Milord* [24], has been used to develop a series of applications in different fields: Medicine, *Pneumon–IA* for the diagnosis of common-acquired pneumonia [26, 27], *Renoir* for the diagnosis of arthropaties [7, 17, 8]; Biology, *Spong–IA* for the classification of north-atlanto-mediterranean sponges [10, 9]; Agriculture, *Gtep–IA* for the management of pig farms [19]. Other applications are currently being developed. We think it is worth looking at the references describing applications to get a much clearer idea of the *use* of the language in different domains.

There is a major component on a language description that has been deliberately omitted here, that is, the semantics of the language. An initial approach to its semantics can be found in [22] but a definite one is still not finished and will merit another paper with more technical contents in the near future. This paper concentrates mainly on the syntax description and the operational view of how a **Milord II** program executes.

Milord II has been developed in Common Lisp (the interpreter) and in C (the compiler). This software is available for research and educational purposes. A fresh version for Macintosh[12] machines can be obtained by *anonymous ftp* at ftp.iiia.csic.es in the directory Milord/mac. You can also find those versions and more information on **Milord II** in the WWW at http://www.iiia.csic.es/~milord. Versions for PC and Unix environments will be produced in short time.

Acknowledgements

We are gratefully indebted to the Spanish Comisión Interministerial de Ciencia y Tecnología (CICYT) for its continuated support to the development of this language through the projects ACRE (CAYCIT 836/86), SPES (880J382), ARREL (TIC92–0579–C02–01) and SMASH (TIC96–1038–C04–01). This research has also been supported by Esprit Basic Research Action number 3085 (DRUMS) and by the European Community project MUM (Copernicus 10053). The definition of this

[12] Macintosh is a trademark of Apple Computer, Inc.

language has profited from ideas coming from many people, among which we would like to thank Ramón López de Mántaras, Lluís Godo, Francesc Esteva, Jaume Agustí and Don Sannella. Our thanks to the people that have developed applications based on **Milord II**, Albert Verdaguer, Miquel Belmonte, Marta Domingo, Pilar Barrufet, Lluís Murgui and Ferran Sanz.

The final version of the paper has been produced while Carles Sierra was on sabbatical leave at the Queen Mary and Westfield College, University of London, thanks to the Spanish Ministry of Education grant PR95–313.

References

[1] J. Agustí, F. Esteva, P. Garcia, L. Godo, R. Lopez de Mantaras, and C. Sierra. Local multi-valued logics in modular expert systems. *Journal of Experimental and Theoretical Artificial Intelligence*, 6(3):303–321, 1994.

[2] J. Agustí, F. Esteva, P. Garcia, L. Godo, Ramón López de Mántaras, J. Puyol, C. Sierra, and L. Murgui. *Fuzzy Logic for the Management of Uncertainty*, chapter Structured Local Fuzzy Logics in Milord, pages 523–551. John Wiley and Sons, Inc., 1992.

[3] J. Agustí, J. Esteva, P. Garcia, L. Godo, and C. Sierra. Combining multiple–valued logics in modular expert systems. In *Proceedings 7th Conference on Uncertainty in AI*, 1991.

[4] J. Agustí, C. Sierra, and D. Sannella. *Methodologies for Intelligent Systems, 4*, chapter Adding generic modules to flat rule–based languages: A low cost approach, pages 43–51. Elsevier Science Publishing Co., Inc., 1989.

[5] J. L. Arcos. Definició i implementació d'un compilador per a Milord II. Master's thesis, Universitat Politècnica de Catalunya, Barcelona, 1992.

[6] Pilar Barrufet. Terap-ia: Sistema expert d'ajuda a la presa de decisions en el tractament antibiòtic i en l'indicació d'hospitalització de les pneumònies extrahospitalaries en el adult. Technical Report IIIA–RR–96–18, IIIA–CSIC, 1996.

[7] M. Belmonte. *Renoir: Un sistema experto para la ayuda en el diagnostico de colagenosis y artropatias inflamatorias.* PhD thesis, Universitat Autònoma de Barcelona, 1991.

[8] Miquel Belmonte, Carles Sierra, and Ramón López de Mántaras. Renoir: An expert system using fuzzy logic for rheumatology diagnosis. *International Journal of Intelligent Systems*, 9(11):985–1000, 1994.

[9] M. Domingo and C. Sierra. A knowledge level analysis of taxonomic domains. In press. International Journal of Intelligent Systems, 1996.

[10] Marta Domingo. *An expert system architecture for taxonomic domains. An application in Porifera: the development of Spongia.* PhD thesis, Universitat de Barcelona, 1995.

[11] Marta Domingo. *An Expert System Architecture for Identification in Biology*, volume 4 of *Monografies del IIIA*. IIIA–CSIC, 1996.

[12] F. Esteva, P. Garcia-Calves, and L. Godo. Enriched interval bilattices: An approach to deal with uncertainty and imprecision. *Uncertainty, Fuzzyness and Knowledge–Based Systems*, 1994.

[13] L. Godo, Ramón López de Mántaras, C. Sierra, and A. Verdaguer. Managing linguistically expressed uncertainty in milord application to medical diagnosis. *AI Communication*, 1(1):14–31, 1988.

[14] L. Godo, Ramón López de Mántaras, C. Sierra, and A. Verdaguer. Milord: The architecture and management of linguistically expressed uncertainty. *International Journal of Intelligent Systems*, 4:471–501, 1989.

[15] L. Godo and C. Sierra. On knowledge base refinement. In *Proceedings of the 14th IMACS World Congress on Computational and Applied Mathematics. Atlanta IMACS'94*, volume 3, pages 1477–1479, 1994. Invited paper.

[16] R. Harper, D. McQueen, and R. Milner. Standard ML. Technical Report ECS–LCFS–86–2, Edinburgh University, 1986.

[17] Ch. Hernandez, J.J. Sancho, Miquel Belmonte, Carles Sierra, and Ferran Sanz. Validation of the medical expert system renoir. *Computers and Biomedical Research*, 27(6):441–471, 1994.

[18] Ramón López de Mántaras. *Approximate Reasoning Models.* Ellis Horwood Series in Artificial Intelligence, 1990.

[19] Jesús Pomar, Marta Domingo, and Josep Lluís Noguera. *Information and Communication Technology Applications in Agriculture*, volume 10 of *Agro-Informaticsreeks*, chapter Fuzzy Reasoning Knowledge-Based Systems for Pig Management, pages 287–291. Wageningen (Nederlands), 1996.

[20] J. Puyol, L. Godo, and C. Sierra. A specialisation calculus to improve expert system communication. In *Proceedings ECAI'92*, pages 144–148, 1992.

[21] Josep Puyol-Gruart. *Modularization, Uncertainty, Reflective Control and Deduction by Specialization in MILORD II, a Language for Knowledge–Based Systems.* PhD thesis, Universitat Autònoma de Barcelona, 1994.

[22] Josep Puyol-Gruart. *MILORD II: A Language for Knowledge–Based Systems*, volume 1 of *Monografies del IIIA*. IIIA–CSIC, 1996.

[23] E. H. Shortliffe. *Computer Based Medical Consultations: MYCIN.* American Elsevier, New York, 1976.

[24] C. Sierra. *MILORD: Arquitectura multi–nivell per a sistemes experts en classificació.* PhD thesis, Universitat Politècnica de Catalunya, Barcelona, 1989.

[25] Guy Steele. *Common Lisp: The Language.* Digital Press, 1984.

[26] A. Verdaguer. *Pneumon–IA: Desenvolupament i validació d'un sistema expert d'ajuda al diagnòstic mèdic.* PhD thesis, Universitat Autònoma de Barcelona, 1989.

[27] Albert Verdaguer, A. Patak, J.J. Sancho, Carles Sierra, and Ferran Sanz. Validation of the medical expert system pneumonia. *Computers and Biomedical Research*, 25(6):511–526, 1992.

[28] Lluís Vila. *On Temporal Representation and Reasoning in Knowledge–Based Systems*, volume 3 of *Monografies del IIIA*. IIIA–CSIC, 1996.

A Syntax

PROGRAM ::= moddecl$^+$
moddecl ::= **Module** *amodid* [([paramlist])] [modoper modexpr] [= modexpr]
paramlist ::= paramlist ; paramlist | *amodid* modoper modexpr
modoper ::= : | ≥ | ≤
modexpr ::= bodyexpr modoper modexpr | bodyexpr
bodyexpr ::= **begin** decl **end** | pathid [([iparamlist])]
iparamlist ::= modexpr ; iparamlist | modexpr
pathid ::= *amodid* | *amodid*/pathid
decl ::= [hierarchy] [interface] [deductive] [control]
hierarchy ::= moddecl | **Inherit** *modid* | **Open** bodyexpr | hierarchy hierarchy
interface ::= [**Import** predicateidlist] [**Export** predicateidlist]
predicateidlist ::= *predid* , predicateidlist | *predid*
deductive ::= **Deductive knowledge**
[**Dictionary:** [**Types:** typebinding$^+$] **Predicates:** predicate$^+$]
[**Rules:** rule$^+$]
[**Inference system:** logcomp]
end deductive
typebinding ::= *typeid* [= typespec]
typespec ::= **boolean** | **many-valued** | **numeric** | **class** |
fuzzy char-funct | (symbollist) | (valuesspec) | *typeid*
symbollist ::= *symbol* [*string*] | symbollist , symbollist
valuesspec ::= *symbol* [*string*] char-funct | valuesspec , valuesspec
char-funct ::= (*number* , *number* , *number* , *number*)
predicate ::= *predid* = attributes
attributes ::= [name] [question] type [function] [relation$^+$] [explanation] [image]
name ::= **Name:** *string*
question ::= **Question:** *string*
type ::= **Type:** typespec
function ::= **Function:** (S-expression)
S-expression ::= *atom* | list | predef-func | S-expression S-expression
list ::= (S-expression) | ( )
predef-func ::= (**Type** *predid*) | (**Linguistic_terms**)
relation ::= **Relation:** relationid pathpredid
pathpredid ::= pathid/*predid* | *predid*
relationid ::= **Needs** | **Needs_true** | **Needs_false** | **Needs_value** |
Belongs_to | **Needs_quantitative** | **Needs_qualitative** | *symbol*
explanation ::= **Explanation:** *string*
image ::= **Image:** *fileid*
rule ::= *ruleid* **If** premisse-rule **Then** conclusion-rule [*documentation*]
premisse-rule ::= condition-rule **and** premisse-rule | condition-rule
condition-rule ::= conditio | **no** (conditio)
conditio ::= operator (expression, ..., expression) |
expression operator expression |
pathpredid | *ltermid* | **true** | **false**
expression ::= operator-arit (expression, ..., expression) |
(expression operator-arit expression) |

number | pathpredid

operator ::= < | > | <= | >= | = | /= | **int** |

operator-arit ::= + | - | * | :

conclusion-rule ::= **conclude** rconclusion **is** cert-value

rconclusion ::= *predid* | *predid* = *symbol* | **no** (*predid*) | **no** (*predid* = *symbol*)

logcomp ::= [lingtermdef] [conjunction] [renaming]

lingtermdef ::= **Truth values** = (ltermidlist)

ltermidlist ::= *ltermid* , ltermidlist | *ltermid*

conjunction ::= **Conjunction** = truth-table

truth-table ::= **Truth table** (arrows)

arrows ::= (ltermid$^+$) | arrows arrows

renaming ::= **Renaming** lrenames$^+$

lrenames ::= pathid/*ltermid* ==> cert-value

cert-value ::= *ltermid* | [*ltermid* , *ltermid*]

control ::= **Control knowledge**
[**Evaluation type:** evaltype]
[**Truth threshold:** *ltermid*]
[deduccnt] [structcnt]
end control

evaltype ::= **lazy** | **eager** | **reified**

deduccnt ::= **Deductive control:** mrr$^+$

mrr ::= metaid **If** premisse-meta **Then** filter-mrr$^+$

premisse-meta ::= mexpr **and** premisse-meta | mexpr

filter-mrr ::= **inhibit rules** [relation-id] pathpredid |
prune pathpredid | **conclude** gexpr | **conclude** known

structcnt ::= **Structural control:** mre$^+$

mre ::= metaid **If** premisse-meta **Then** filter-mre

filter-mre ::= **filter** amodid$^+$ |
order amodid$^+$ **with certainty** cert-value |
Open (term) | **Module** (term) | **Inherit** (*amodid*)

mrx ::= metaid **If** premisse-meta **Then** exception

exception ::= **definitive solution** predid |
stop

mexpr ::= known | mrel | msubmod | mthres | card | atom | member |
eqdif | moper | int | setof | pos | gexpr

symorvar ::= *symbol* | **$***symbol*

vpath ::= symorvar | symorvar/vpath

known ::= **K** (fact, interval)

fact ::= factex | **not**(factex) | **implies**(list,list)

factex ::= vpath | =(vpath, symorvar)

interval ::= **$***symbol* | **int**(symorvar, symorvar)

mrel ::= *relationid*(symorvar, vpath)

msubmod ::= **submodule**(symorvar, symorvar) | **submodule**(symorvar)

mthres ::= **threshold**(symorvar, symorvar) | **threshold**(symorvar)

card ::= **cardinal**(list, symorvar)

list ::= **$***listid* | (listelem)

listelem ::= *elemid* | *elemid* , listelem

atom ::= **atom**(list)

member ::= **member** (symorvar, list)
eqdif ::= **equal(** listorsym,listorsym) | **diff(** listorsym,listorsym)
listorsym ::= symbol | list
moper ::= loper(symorvar,symorvar)
loper ::= **lt** | **le** | **eq** | **neq** | **ge** | **gt**
int ::= **intersection(** list,list)
setof ::= **set_of_instances** (**$***var*,term,**$***var*)
pos ::= **position(** symorvar,list,symorvar)
gexpr ::= *predid* (term , ... , term)
term ::= **$***varid* | *symbol* | *termid*(term, ..., term)

B Default Logic

This is the default logic used in **Milord II** when there is no local logic declaration into a module.

Truth-values

$$A_8 = \{gp, mpop, llp, modp, p, fp, mp, s\}$$

where the meaning of each term is the following:

1. `gp`: Impossible
2. `mpop`: Very few possible
3. `llp`: Few possible
4. `modp`: Slightly possible
5. `p`: Possible
6. `fp`: Quite possible
7. `mp`: Very possible
8. `s`: Definite

Conjunction This operation is described in the Table 3.

T_8	gp	mpop	llp	modp	p	fp	mp	s
gp	gp	gp	gp	gp	gp	gp	gp	gp
mpop	gp	mpop	mpop	mpop	mpop	mpop	mpop	mpop
llp	gp	mpop	mpop	llp	llp	llp	llp	llp
modp	gp	mpop	llp	modp	modp	modp	modp	modp
p	gp	mpop	llp	modp	modp	modp	p	p
fp	gp	mpop	llp	modp	modp	p	fp	fp
mp	gp	mpop	llp	modp	p	fp	mp	mp
s	gp	mpop	llp	modp	p	fp	mp	s

Table 3: Conjunction table for A_8.