HANDBOOK OF DEFEASIBLE REASONING AND UNCERTAINTY MANAGEMENT SYSTEMS

Editors:
Dov M. Gabbay and Philippe Smets

VOLUME 7
Agent-based Defeasible Control in Dynamic Environments

Edited by: J.–J. Ch. Meyer and J. Treur

KLUWER ACADEMIC PUBLISHERS

LLUÍS GODO, JOSEP PUYOL-GRUART AND CARLES SIERRA

CONTROL TECHNIQUES FOR COMPLEX REASONING: THE CASE OF *MILORD II*

1 INTRODUCTION

Reasoning patterns occurring in complex problem solving tasks usually cannot be modelled by means of just a pure classical logic approach. This is due to several reasons, for instance: incompleteness of the available information, need of using and representing uncertain or imprecise knowledge, or combinatorial explosion of classical theorem proving when knowledge bases become large. To deal with these problems, *Milord II*, an architecture for Knowledge Base Systems (KBS), combines modularization techniques with both implicit and explicit control mechanisms and with an approximate reasoning component based on many-valued logics.

Roughly speaking, a Knowledge Base (KB) in *Milord II* consists of a hierarchy of modules interconnected by their export interfaces. Each module contains an Object Level Theory (OLT) and a Meta-Level Theory (MLT) interacting through a reflective mechanism (see Figure 1).

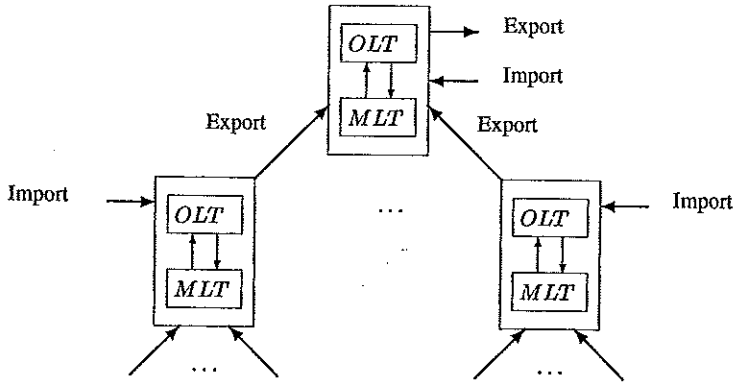

Figure 1. Structure of a *Milord II* module hierarchy.

A module can be understood as a functional abstraction between the set of components it needs as input and the type of results it can produce. From the logical point of view, *Milord II* makes use of both many-valued logic and epistemic

J.J.Ch. Meyer and J. Treur (eds.),
Handbook of Defeasible Reasoning and Uncertainty Management Systems, Vol. 7, 65–97.

meta-predicates to express the truth status of propositions. For further details in these logical topics the reader is referred to [Godo *et al.*, 1995; Puyol *et al.*, 1992; Puyol-Gruart *et al.*, 1998; Puyol-Gruart and Sierra, 1997; Sierra and Godo, 1992; Sierra and Godo, 1993].

In this chapter we focus on the control techniques used in *Milord II* that determine a KBS execution. The explicit part of the control, declarative in nature, is mainly based on a reflective approach and a declarative backtracking mechanism. In this context, reflection makes sense as a control mechanism because there is a clear separation between domain (object-level) and control (meta-level) knowledge. The basic implicit control components are a subsumption mechanism and a process of elimination of unnecessary rules, both concerning the object level.

Next we list the most usual control requirements for a KBS language together with the solutions adopted in *Milord II*.

Locality of Control: All explicit control mechanisms are specified locally to each module. This allows us to identify a module as the complete description of a problem (or subproblem). The separation between domain and control knowledge is a typical characteristic of most KBS languages to offer a clear and declarative programming style.

Specificity versus generality: To solve problems, human experts are able to reason at different levels of precision depending on the amount of data at hand. For instance, a physician cannot always gather all the relevant data to make a complete and accurate diagnosis. This is the case, for example, when a patient is in a coma and thus the physician cannot pose him any question. Nonetheless, the physician has to make a diagnosis, although it may be provisional. To represent these situations *Milord II* provides the knowledge engineer with two different control options:

- To write rules with different levels of specificity (using more or less information, that is, putting more or less conditions) deducing the same conclusion with possibly different levels of belief. To deal with this kind of rules, *Milord II* extends the concept of subsumption by associating sets of *partial labels* to the rules. This technique guarantees the use of the more specific knowledge whenever possible.
- To encode default-like rules (by means of meta-rules) that generate plausible assumptions to be used when a piece of relevant information is missing (see Section 9).

Avoidance of unnecessary work: *Milord II* takes advantage of the specialization deductive mechanism [Puyol *et al.*, 1992; Puyol-Gruart *et al.*, 1998] to eagerly detect when a rule cannot increase the certainty on a conclusion. When a rule is applied, *Milord II*'s engine decides whether other rules with the same conclusion can increase its certainty or not. If not, they are removed.

Locality of threshold: In some cases knowledge engineers are interested in programming modules whose deduced predicates are only useful if their certainty is above a minimum truth level. This is done by declaring a *threshold* local to each module. Whenever a rule gets, by specialization, a truth interval with its minimum value below the module threshold, it is removed.

Flexibility in data gathering: Given a query to a module, different strategies for the module to get an answer can be used. The different evaluation strategies of *Milord II* determine how and in which order the necessary external information is gathered.

Declarativity of Control: *Milord II* Horn-like meta-rules are used as a declarative language to implement several control actions, e.g. elimination of rules, generation of plausible assumptions, dynamic changing of the modules hierarchy or dynamic creation of modules.

The detailed description of the different implicit and explicit control mechanism of *Milord II* is structured in this Chapter as follows. In Section 2 we present a general picture of the whole control structure. Sections 3 through 8 are devoted to describe the different *Milord II* control mechanisms, that is, the object level process, the upwards reflection operation, the meta-level process, the downwards reflection operation and the communication among modules respectively. In Section 9 two reasoning tasks are implemented using some of the previously presented control mechanisms.

2 *MILORD II* OVERVIEW

A *Milord II* KB consists of a hierarchy of modules, each module containing different kinds of knowledge, structured as sketched in Figure 2.

From a logical point of view, a module is composed of an Object Level Theory (OLT) and a Meta Level Theory (MLT). The OLT is generated by a set of rules which are specified in the `Deductive Knowledge` definition. These rules are formulas belonging to the Object Level Language $\mathcal{OL}_n$, a propositional language based on many-valued semantics. Formulas of this language are of the form (r, V), where r is a Horn-like rule and V is an interval of truth values belonging to a finite and totally ordered set of values, also specified in the module declaration. Deduction in the object level language, denoted $\vdash_{\mathcal{O}}$, is mainly based on a *specialization inference rule*, a straightforward generalization of the many-valued version of Modus Ponens, which allows to simplify rules as soon as we know truth-intervals for any of their conditions. On the other hand, the MLT is generated by a set of meta-rules which are specified in the `Deductive Control` definition. These meta-rules are formulas of the Meta Level Language $\mathcal{ML}_n$, a restricted first order classical language of Horn rules. Variables in meta-rules, if any, are considered universally quantified. Deduction at the meta level, denoted by

$\vdash_{\mathcal{M}}$, is based on Modus Ponens and particularization. The overall reasoning process of a module consists on reasoning at each level and interacting between both levels. This process produces a sequence of modifications over the initial OLT and MLT. For a deeper insight of *Milord II* modules, the reader is again referred to [Godo *et al.*, 1995; Sierra and Godo, 1993].

From an operational point of view, a module can be identified with a process attached to it, used to compute values (truth intervals) for all the propositions and variables contained in its export interface. Namely, a module execution consists of the reasoning process necessary to compute the values for the propositions and variables in the export interface the user queries about. The execution of a module can possibly activate the execution of submodules in the hierarchy. These executions only interact with the parent module through the export interface of the submodules, giving formulas back as result. It is worth noticing that the interaction is made only at the object level.

```
Begin
      Hierarchy of submodules
      Import:  ...
      Export:  ...
      Deductive knowledge
           Dictionary:  ...
           Rules:  ...
           Inference System:  ...
                Truth-values:  ...
                Connectives:  ...
                Renaming:  ...
      end deductive
      Control knowledge
           Evaluation Type:  ...
           Truth Threshold:  ...
           Deductive Control:  ...
           Structural Control:  ...
      end control
end
```

Figure 2. Knowledge components of a module.

Conceptually, the execution of a module involves two deductive subprocesses, object and meta-level, that act as co-routines, and three operations. Two of them, *upwards reflection* and *downwards reflection*, are resume-type operations between the co-routines that, besides acting as resuming operations, modify the knowledge used by the deductive subprocesses. The third operation is the *communication* with the user and/or other modules that has the effect of adding new formulas to the object-level co-routine. Figure 3 shows the structure of a module and the relations

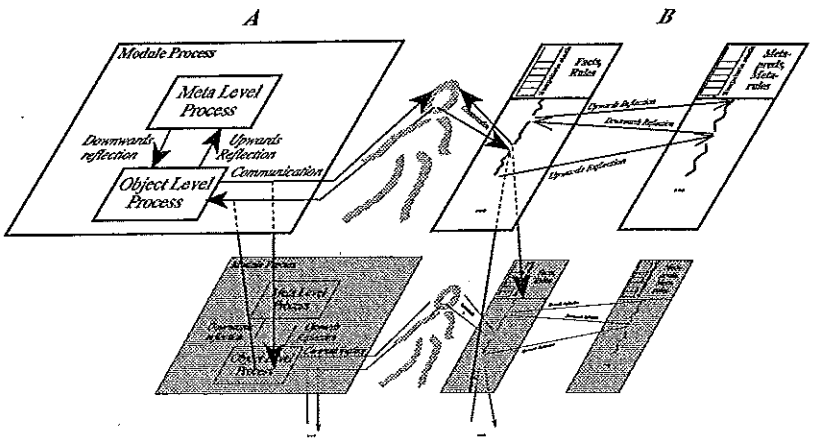

Figure 3. A: structure of the components of *Milord* module process.
B: Co-routine view of a module process.

between its components. Besides that, the module evaluation type determines in which way subprocesses and operations are combined to get the global control behaviour of a module execution.

Next we succinctly describe each one of the above mentioned processes and operations.

Object Level Process: This process uses as data the set of propositional variables and rules of the module possibly updated by the previous downwards reflection and communication operations. With this data and a goal to be solved, the task of the process is to obtain a value for the goal and potentially for other propositional variables of the module. Obtaining a value for a propositional variable can be done in one of the following ways:[1] by using the *communication* operation, either by querying the user (when the propositional variable is declared as *Import*) or querying a submodule (when the propositional variable belongs to the export interface of a submodule); or by *deduction* when the propositional variable is the conclusion of a rule. To do so, the process follows the rule specialization algorithm with two implicit control mechanisms, namely the *subsumption* and the *elimination of unnecessary rules*, and a parametric control mechanism, the *truth-threshold rule elimination*.

The type of evaluation determines when the control is passed to the upwards reflection or to the communication operations.

[1] Actually there are other ways such as functional evaluation—in the case of propositional variables with an attached function—or constraint propagation, but they are out of the scope of this paper.

Upwards Reflection Operation: This operation translates a subset of the current object level formulas in the object process to meta-predicate instances in the meta-level process. Once the operation concludes, the meta-level process is resumed.

Meta Level Process: The meta level process takes as input the set of meta-rules of a module and the set of meta-predicate instances generated by the upwards reflection operation, together with the meta-predicate instances that had been previously deduced. The process then makes use of a forward inference engine with a depth-first control strategy, following the writing order of meta-rules. The stop condition is the impossibility of applying any meta-rule. In that case, the process resumes the object level process through the downwards reflection operation.

Downwards Reflection Operation: This operation is the dual of the upwards reflection one. It translates formulas from the meta-level process into the object level one and executes the actions determined by the meta-level process. When the translation is finished, the object level process is resumed. Special mention has to be made when an instance of the action *Assume* is applied. In this case, as many extensions of the meta-theory MLT as elements in the argument of the *Assume* action are generated (see Figure 4). These extensions conform a tree of MLTs. Every time an *Assume* action is executed a new branching is added to this tree. Whenever a *Resume* action is executed, a backtracking in that tree is performed and the computation is resumed. This is how the declarative backtracking mechanism (see Section 7) is implemented.

Communication: This operation is used to add new formulas to the object-level process either from the external user or from other modules of the KB. The evaluation type determines when this operation is to be applied.

The control mechanisms determine the algorithmic behaviour of the processes themselves or just the way processes and operations are combined. The combination of the previous processes and operations is done by the explicit declaration of the evaluation strategy inside each module. In *Milord II* there are three evaluation strategies: *lazy*, *eager* and *reified*. Each one of them produces a different behaviour. On the one hand, the lazy and eager evaluation types are opposite strategies about how to obtain external data (from the user and/or from its submodules). The lazy strategy always tries to use the minimum information while the eager strategy makes use of as much information as possible.

In the following algorithmic descriptions of the evaluation strategies, OLP stands for object level process and MLP for meta level process.

Lazy: A module with lazy evaluation finds the cheapest path to compute a solution for a goal, that is, no irrelevant data will ever be gathered. The control

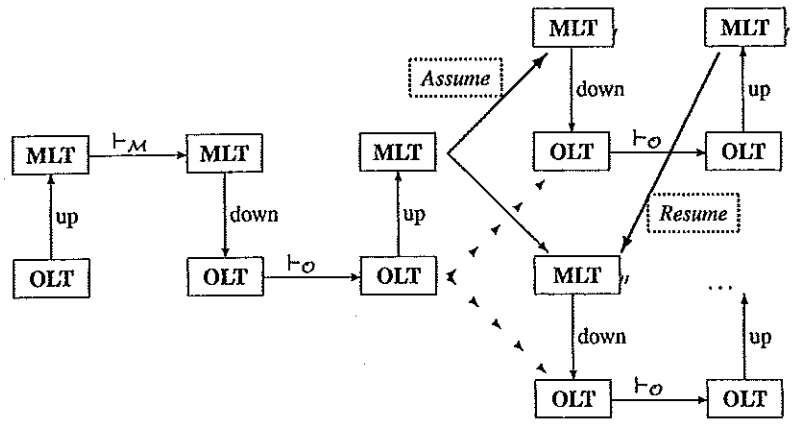

Figure 4. Module processes and operations. *Assume* and *Resume* actions.

used in the module to answer a query, in a simplified view, is a loop over first finding the next relevant propositional variable to look for a value and then specializing the deductive knowledge. This cycle is repeated until the goal is solved or no more relevant questions exist. This is the evaluation strategy used by default.

Given a query to a lazy module, the control flow of the module process is the following one:[2]

1. [OLP] If the goal has already a value, STOP.
2. [OLP] Otherwise, depending on the kind of goal, it performs one of the next steps in order to get a value for it.
   (a) *Submodule goal.* If the goal is a path to a submodule of the current module, and if that submodule is visible,[3] then call the communication operation with this goal (the communication operation will call the submodule object process to solve the goal).
   (b) *Goal belonging to the import interface.* Call the communication operation with this goal (the communication operation will query the user to give a value for the goal).

[2] A symbol between square brackets stands for the name of the active process in which the algorithm step is performed

[3] Submodules can be hidden by a refinement operation between modules [Godo and Sierra, 1994]. This kind of operation is out of the scope of this paper.

(c) *Goal with a function attribute.* Now the evaluation of the goal depends on the evaluation of the function associated to the propositional variable. If there are arguments of the function with no value, call recursively the lazy algorithm over them from left to right. When all arguments have values, evaluate the function.

(d) *Goal that can be deduced by means of rules.* In this case we start a depth-first search on the rules of the module deducing the goal to look for a propositional variable without value. The search algorithm orders rules according to the following criteria, in order of preference.

i. *More specific rules first.* We try to find solutions by first using the more specific rules—those with less conditions.

ii. *More precise rules first.* A rule is more precise than another when its truth-value interval is more precise. Notice that this order can change during the execution because of the specialization of rules.

iii. *The writing order of rules.*

To evaluate the conditions of a selected rule, the search strategy follows the writing order of the conditions (left to right), in a depth-first manner. Finally, call recursively the lazy algorithm with the above mentioned propositional variable as a subgoal.

Notice that the algorithm finally returns a path to a submodule of the current module, a propositional variable belonging to its import interface, or a propositional variable with a evaluable function associated to it, and its associated value.

3. [OLP] Specialization of rules
4. [OLP] Call the reification operation
5. [MLP] The meta level fires all possible meta-rules.
6. [MLP] Call the reflection operation
7. [OLP] If the reflection operation does not modify the object level set of formulas GOTO 1, otherwise GOTO 3.

Notice that this algorithm always provides the goal with a value since, in the worst case, it will get the value *unknown*, which corresponds to the maximum imprecision interval.

Eager: An eager strategy asks the user for all the variables and propositions declared in the *Import* interface of the module and queries all the exportable propositional variables of its submodules as well.

Given a query to a module with an eager evaluation, the control flow of the module process is the following one:

1. [OLP] If the goal has already a value, STOP.
2. [OLP] Otherwise, call the communication operation as many times as necessary to get values for all imported propositional variables in their writing order.
3. Steps 3, 4, 5 and 6 of the Lazy evaluation algorithm.
4. FOR each submodule DO (Submodules are ordered by their writing order).
   (a) [OLP] call the communication operation to get values for all the submodule exportable propositional variables used in the rules or meta-rules of the module.
   (b) Steps 3, 4, 5 and 6 of the Lazy evaluation algorithm.

   END FOR
5. [OLP] If the goal has already a value, STOP.
6. Steps 3, 4, 5 and 6 of the Lazy evaluation algorithm.
7. [OLP] GOTO 4

Reified: This kind of evaluation strategy does not differ from the eager one in the way of gathering data. The main difference of a reified strategy with respect to both lazy and eager strategies is that the specialization mechanism of the object level is not used at all. Therefore, deduction is only performed at the meta-level process. The motivation behind this evaluation strategy is to provide module designers with the possibility to define meta-interpreters.

3 OBJECT LEVEL PROCESS

3.1 *Object-level deduction*

Milord II provides the user with approximate reasoning capabilities at the object level. The approximate reasoning mechanisms are based on the use of a finitely-valued fuzzy (or many-valued) logic. Before describing the logical deduction system, and for the sake of a better understanding, we first outline the semantics behind it.

A particular many-valued logic can be specified inside each module by defining which is the algebra of truth-values, i.e. which is the (finite) ordered set of truth-values and which is the set of logical operators associated to them. Formally speaking, a *Milord II* algebra of truth-values $A_{n,T} = \langle A_n, \leq, N_n, T, I_T \rangle$ is a finite linearly ordered residuated lattice with a negation operation. In plain words, the set of truth-values $A_n = \{0 = a_1 < a_2 < ... < a_n = 1\}$ is a chain of n elements where 0 and 1 are the booleans *false* and *true* respectively; the negation operation N_n is the involution in A_n, i.e. $N_n(a_i) = a_{n-i+1}$; the conjunction operator T is a t-norm, i.e. a binary, commutative, associative and

non-decreasing operation on A_n with **1** as neutral element and **0** as null element; finally I_T is the residuum of T, i.e. defined as $I_T(a,b) = Max\{c \in A_n \mid T(a,c) \leq b\}$, and it is used to model a many-valued implication. As it is easy to notice from the above definition, any of such truth-values algebras is completely determined as soon as the set of truth-values A_n and the conjunction operator T are chosen. So, varying these two characteristics we generate a family of different multiple-valued logics. For instance, taking $T(a_i,a_j) = a_{min(i,j)}$ or $T(a_i,a_j) = a_{min(n,n-i+j)}$ we get the well-known Gödel's and Łukasiewicz's semantics (truth-tables) for finitely-valued logics [Gottwald, 1988; Gottwald, 1993; Hájek, 1998; Hájek, 1995].

In a given module, and thus for a given truth-value algebra A_n and a set of propositional variables $\Sigma_{\mathcal{O}}$, the set $\mathcal{OL}_n$ of object-level formulas consists of:

- $\mathcal{OL}_n$-Atoms: $\{(p,V) \mid p \in \Sigma_{\mathcal{O}}\}$
- $\mathcal{OL}_n$-Literals: $\{(p,V),(\neg p,V) \mid (p,V) \in \mathcal{OL}\text{-Atoms}\}$
- $\mathcal{OL}_n$-Rules: $\{(p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow q, V^*) \mid p_i$ and q are literals (atoms or negations of atoms) and $\forall i,j(p_i \neq p_j, p_i \neq \neg p_j, q \neq p_j, q \neq \neg p_j)\}$

where V and V^* are intervals of truth-values. Intervals V^* for rules are constrained to be upper intervals, i.e. of the form $[a,1]$, where $a > 0$. That is, object level formulas are indeed signed formulas under the form of pairs of usual propositional formulas (restricted to be literals or rules) and intervals of truth-values.

The *semantics* is obviously determined by the connective operators of the truth-value algebra $A_{n,T}$. *Interpretations* are defined by valuations ρ mapping the (propositional) sentences to truth-values of A_n fulfilling the following conditions:[4]

$\rho(true) = \mathbf{1}$,
$\rho(\neg p) = N_n(\rho(p))$,
$\rho(p_1 \wedge \ldots \wedge p_n \rightarrow q) = I_T(T(\rho(p_1),\ldots,\rho(p_n)),\rho(q))$.

Then the *satisfaction relation* between interpretations and $\mathcal{OL}_n$-formulas is defined as

$$\rho \models_{\mathcal{O}} (\varphi, V) \text{ iff } \rho(\varphi) \in V$$

and it is extended to a *semantical entailment* between sets of $\mathcal{OL}_n$-formulas and $\mathcal{OL}_n$-formulas as usual:

$$\Gamma \models_{\mathcal{O}} (\varphi, V) \text{ iff } \rho \models_{\mathcal{O}} (\varphi, V) \text{ for all } \rho \text{ such that } \rho \models_{\mathcal{O}} A, \text{ for all } A \in \Gamma.$$

Once the semantics is clear, we come to the (syntactical) deduction system which is implemented in each module. The *Many-valued Specialisation Calculus* (**Mv-SC** for short) is defined by the following axioms:

- **A1**: $(\varphi,[0,1])$

[4]The expression $T(r_1,r_2,r_3,\ldots)$ is the recurrent application of T as $T(r_1,T(r_2,T(r_3\ldots)))$.

- **A2**: $(true,1)$

and by the following inference rules:

- **Weakening**: from (φ,V_1) infer (φ,V_2), where $V_1 \subseteq V_2$
- **Not-introduction**: from (p,V) infer $(\neg p, N_n^*(V))$
- **Not-elimination**: from $(\neg p,V)$ infer $(p,N_n^*(V))$
- **Composition**: from (φ,V_1) and (φ,V_2) infer $(\varphi,V_1 \cap V_2)$
- **Specialization**: from (p_i,V) and $(p_1 \wedge \cdots \wedge p_n \rightarrow q, W^*)$ infer $(p_1 \wedge \cdots \wedge p_{i-1} \wedge p_{i+1} \wedge \cdots \wedge p_n \rightarrow q, MP_T^*(V,W^*))$

where $N_n^*([a,b]) = [N_n(b),N_n(a)]$ is the point-wise extension of N_n to intervals and $MP_T^*(V,W^*)$ is defined as follows: $MP_T^*([a,b],[c,1]) = [T(a,c),1]$. In [Puyol-Gruart *et al.*, 1998] it is shown that this deductive system is sound with respect to the above semantics and complete for deriving $\mathcal{OL}_n$-atoms. Object-level deduction will be denoted by $\vdash_{\mathcal{O}}$.

3.2 *Object Level Control Mechanisms*

Subsumption mechanism

When expressing the deductive knowledge of a module, experts might write different rules concluding the same propositional variable to represent the possibility of either:

- having different unrelated sets of conditions entailing that propositional variable, or
- having different sets of conditions related by an inclusion relation that may allow concluding that propositional variable (with different certainty values). Subsumption is the mechanism that ensures that only the most specific sets of conditions will be used.

The widely accepted subsumption criterion is to use always the more specific knowledge in the deductive process. This idea is made precise in the following general definition.

DEFINITION 1 (Subsumption). Given a knowledge base KB and two rules $R_1 : (A_1 \rightarrow B, \alpha_1)$ and $R_2 : (A_2 \rightarrow C, \alpha_2)$ where B and C are literals over the same propositional variable, we say that rule R_1 is more specific than rule R_2 if, taking for granted the set of formulas in KB, whenever A_1 is true A_2 is also true, that is, when

$$KB \models A_1 \rightarrow A_2.$$

In the particular multi-valued logical framework of *Milord II* this definition can be expressed as $\rho(A_1 \to A_2) = 1$, for all many-valued interpretation ρ such that $\rho \models KB$. By definition of the implication connective as a residuum, the condition $\rho(A_1 \to A_2) = 1$ is equivalently expressed as $\rho(A_1) \leq \rho(A_2)$.

This criterion can be described in terms of the set of *labels*—non deducible propositional variables needed to apply the rule—associated to each premise. Namely, it can be checked that, in the conditions of the above definition, rule R_1 is more specific than rule R_2 if for each label L_i of A_1 there is a label L_j of A_2 such that $L_i \to L_j$ is a valid formula. The condition $\models L_i \to L_j$ reduces to the inclusionship of labels $L_j \subset L_i$.

For instance consider the following set of rules:

$$\begin{cases} R_1 : a \wedge b \wedge c \wedge d \to g \\ R_2 : e \wedge f \to g \\ R_3 : c \to e \\ R_4 : a \wedge b \to f \end{cases}$$

It is easy to see that there is a subsumption relation between the rules R_1 and R_2. The set of non deducible propositional variables necessary to apply the rule R_1 is $\{a, b, c, d\}$, whilst for the rule R_2 is $\{a, b, c\}$. Therefore R_1 is more specific than R_2.

Due to the special deductive mechanism of *Milord II*, based on specialization of rules, the subsumption relation changes as deduction progresses. This is so because the specialization mechanism reduces the conditions in the premises of rules, and thus modifies the reference KB used to compute labels. Because of that, *Milord II* incorporates an algorithm that dynamically computes and completes *partial labels*, in the sense that, the set of labels can be incomplete and even labels may be incomplete.

Elimination of Unnecessary Rules

The maximum precision given to the conclusion of a rule is limited by the truth interval of the rule. Consider a rule with certainty value $[a_r, 1]$ and whose premise has been evaluated to the interval $[a_i, a_j]$. Then, the interval associated to the concluded propositional variable by the application of this rule is given by

$$MP_T^*([a_i, a_j], [a_r, 1]) = [T(a_i, a_r), 1)] = [a'_r, 1] \text{ ,where } a'_r \leq a_r.$$

This consideration leads us to the following definition.

DEFINITION 2. A rule $(A \to q, [a_r, 1])$ is *unnecessary* for a propositional variable $(q, [a_i, a_j])$ if $a_r \leq a_i$. Similarly, a rule $(A \to \neg q, [a_r, 1])$ is *unnecessary* for a propositional variable $(q, [a_i, a_j])$ if $a_r \leq N_n(a_j)$, where N_n is the negation operator.

Therefore we can easily test whether the remaining rules concluding a propositional variable are still useful or not. This is what we call the *elimination of unnecessary rules process*. The test is applied every time a rule is specialized since the specialization mechanism broadens rule intervals. This control technique allows us to save unnecessary deductions as well as unnecessary information requirements and processing.

4 META-LEVEL PROCESS

4.1 *Meta-level deduction*

The meta-level language $\mathcal{ML}_n$, corresponding to an object-level language $\mathcal{OL}_n$, is a restricted classical first order language. It is defined from a set Σ_{rel} of predicate symbols including predicates K, P and WK which play a special role in the reflection mechanism; a set Σ_{act} of action symbols (*inhibit_rules*, *assume*, *resume*, *filter*, *stop* and *module*); a set Σ_{fun} of classical arithmetic function symbols; a set Σ_{con} of constants including the truth-values of A_n and object propositional variables of $\Sigma_{\mathcal{O}}$; and a set Σ_{var} of variable symbols,[5] which can be empty.

Meta-level formulas are either ground literals, in a classical sense, or rules of the type

$$\{P_1 \wedge P_2 \wedge \ldots \wedge P_n \to Q \mid P_i, Q \text{ literals}\},$$

where each variable occurring in Q must occur also in some P_i. Variables in meta-rules, if any, are considered universally quantified. Quantifiers are all outermost. Only the conclusion Q may contain action symbols.

The *semantics* of the language is the classical of first order logic. The meaning of the special predicates K, P and WK will be explained in the next subsection along with the definition of the reification rules which use them to represent object-level sentences.

Finally, the *deduction* system is based on only one (modus ponens-like) inference rule:

$$\text{from } \{P_1 \wedge P_2 \wedge \ldots \wedge P_n \to Q, P'_1, P'_2, \ldots, P'_n\} \text{ infer } Q'$$

where $P'_1, \ldots, P'_n$ are ground instances of $P_1, \ldots, P_n$ respectively, such that there exists a unifier σ for $\{P_1 \wedge P_2 \wedge \ldots \wedge P_n, P'_1 \wedge P'_2 \wedge \ldots \wedge P'_n\}$, and $Q' = \sigma Q$ is the ground instance of Q resulting from σ. The deductive system of *Milord II* meta-level is thus not complete with respect to the classical semantics we use for it. Nevertheless, the deduction mechanism based on this single inference rule is powerful enough for our modelling purposes. Meta-level Deduction will be denoted by the symbol $\vdash_{\mathcal{M}}$.

4.2 *Control Actions*

Control actions may affect the deductive knowledge of a module by inhibiting rules and by branching and backtracking the reasoning process. Control actions

[5] When using *Milord II* syntax variable are prefixed by $, for instance $\$x$.

may also modify the hierarchy of a module by inhibiting modules, or creating new ones. They can also abort the execution.

Inhibit Rules: This action takes out of the OLT a particular set of rules. When we execute `inhibit_rules(pathpredid)`, all the rules containing the propositional variable *pathpredid* in their premises are removed. We can also inhibit all rules containing in their premises propositional variables related to a given one.

Assume: The argument of this action stands for an ordered list of possible assumptions to be made at the object level that can be retracted later on.

Resume: It retracts the latest assumption performed.

Filter: This action consists on inhibiting (filtering) a set of submodules of the module. This means that all the propositions p exported by the filtered submodules will be considered as being $(p, unknown)$

Stop: This is an abort action. In some cases it is necessary to abort the execution when an unrecoverable situation holds.

Module: When a meta-rule concludes an instance of *module*, for example *module(= (A, B))*, an action will be performed, at downwards reflection time, to add a submodule named A and equal to B as the last, in writing order, of the already existing submodules. B can be any allowed modular expression, in particular, the application of a generic module. Generic modules containing as control knowledge meta-rule calls to themselves are allowed. This is the way recursion can be defined inside *Milord II* [Puyol-Gruart and Sierra, 1997].

Assume and *Resume* predicates deserve special attention because they allow to define a backtracking mechanism in *Milord II* (see Section 7), useful to model hypothetical reasoning.

5 UPWARDS REFLECTION OPERATION

The upwards reflection operation translates formulas from the current OLT to the MLT in the form of meta-predicate instances. It relates a sub-theory of the OLT with the set of ground literals of the meta-language ML. The meta-predicate WK is used to relate the set of object mv-literals with the set of ground meta-literals. Given that the constant names used in the MLT are exactly the same as those used in the OLT as proposition names, the quoting functions for literals are omitted for the sake of simplicity. The same applies for the intervals of truth-values. So we will write $WK(p, V)$ instead of $WK(\lceil p \rceil, \lceil V \rceil)$. The reification rules are:

$$\frac{(p, V) \in OLT}{\vdash_{\mathcal{M}} WK(p, V)}$$

$$\frac{(p, V) \notin OLT}{\vdash_{\mathcal{M}} \neg WK(p, V)}$$

$$\frac{(p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow q, V) \in OLT}{\vdash_{\mathcal{M}} WK(implies(and(p_1, p_2, \cdots, p_n), q), V)}$$

The other two meta-predicates, K and P, used in the meta-level language to represent the OLT state are definable from the meta-predicate WK:

$$K(p, [a_i, a_j]) \equiv WK(p, [a_i, a_j]) \wedge \neg WK(p, [a_{i+1}, a_j]) \wedge \neg WK(p, [a_i, a_{j-1}])$$

$$P(p) \equiv WK(p, [a_2, 1])$$

5.1 *Other meta-predicates*

Upwards reflection also contains programmer defined relations between propositional variables, the threshold and the rules. Although the submodules of a module are not persistent, the initial submodules are also reified, as well as those that have been filtered.

Relations: When declaring a propositional variable in *Milord II*, it is possible to establish a relation with another propositional variable, in the same module or in a submodule. There is a set of system-defined relations used for control. Other relations are domain dependent and defined by programmers. The name of the relation used in the definition of propositional variables, corresponds to a binary meta-predicate identifier. The two arguments correspond to the name of the propositional variables being related. For instance the definition

```
p1 = name: ...
     ...
     relation: relationid p2
```

becomes the next meta-predicate instance: $relationid(p_1, p_2)$

Threshold: A certainty threshold is treated as a meta-predicate instance. There is an instance per module and one per each submodule: $threshold(a_i)$ and $threshold(submodule_j, a_k)$.

Submodules: There is a meta-predicate called *submodule* which has an instance per submodule (meta-predicate with only one argument), and an instance per sub-submodule (the same meta-predicate name but with two arguments), that is, $submodule(submodule_1)$ or $submodule(submodule_1, subsubmodule_2)$.

Filtered: Instances of this meta-predicate represent the submodules that have been filtered (removed) by meta-rules, $filtered(submodule)$.

6 DOWNWARDS REFLECTION OPERATION

The downwards reflection operation is responsible of making effective at the object level the consequences of the deduced meta-predicate instances.

$$\frac{K(p,V) \in MLT}{\vdash_O (p,V)}$$

The reflection operation modifies the data structure of the OLP to make it causally connected, using the terminology of Patty Maes [Maes, 1988], with the meta-predicate instances.

7 DECLARATIVE BACKTRACKING

When a meta-rule with an *Assume* action in its conclusion is applied, as many extensions of the meta-theory MLT as elements in the argument set of the *Assume* action are generated. For instance, consider the case where, in a certain moment, we have in the current object theory only the literal

$$(p,1)$$

and MLT consists of the following meta-rule:

$$\text{If } K(p,1) \text{ and } \neg P(q) \text{ then } Assume(\{(q,1),(q,0)\})$$

Suppose also that q could not be proved in OLT. Then, after the upwards reflection process, the current MLT will be the extension of the previous one with the ground literals $K(p,1)$ and $\neg P(q)$. So, now the above meta-rule can be applied, and this causes the system to obtain the conclusion $Assume(\{(q,1),(q,0)\})$. The meaning of the action *Assume* is that the elements of its argument should be assumed in different extensions of the current OLT. This is done by building a tree of MLTs, each containing a K meta-predicate instance for all the elements of the argument of *Assume*, implemented by a snapshots stack. So in this case we obtain the following two different extensions of the current MLT in Figure 5:

$$MLT_1 = MLT \cup K(q,1)$$

$$MLT_2 = MLT \cup K(q,0)$$

From now on, and until another instance of an *Assume* or *Resume* action is obtained, the existing communication (upward reflection and downward reflection) between OLT and MLT is moved to a communication between OLT and MLT_1.[6] Thus, in this case, after the downward reflection process OLT is extended with $(q,1)$.

[6]Computationally speaking, MLT_1 is managed by the same co-routine of MLT but with a new snapshot in the stack containing the state of MLT plus $Assume(\{(q,1),(q,0)\})$.

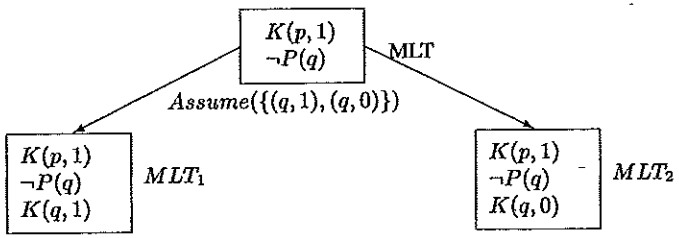

Figure 5. Meta-theories branching using the *Assume* action.

In order to backtrack in the tree of MLTs generated by successive applications of *Assume* actions, the language provides a special 0-ary predicate *Resume*. When a meta-rule concluding *Resume* is applied, we perform a backtracking in the meta-theories tree. This backtracking restores the parent MLT, and the current OLT becomes the OLT which was active at the moment the assumptions were made by the parent MLT. In the above example, backtracking from MLT_1 to MLT makes that q will not be true in the current OLT, and that immediately the communication (upward reflection and downward reflection) between OLT and MLT is moved to a communication between OLT and MLT_2 (see Figure 6).

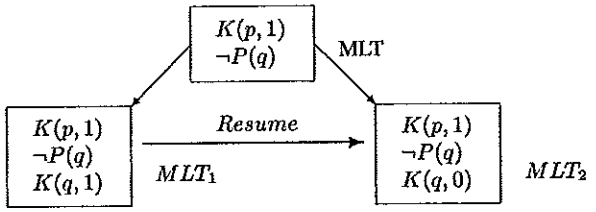

Figure 6. Backtracking using the action *Resume*.

It is worth noticing that actions *Assume* and *Resume* provide the system with a declarative backtracking mechanism, similar to the approach taken in MetaProlog [Bacha, 1988]. This declarative mechanism allows us to implement several complex reasoning patterns. For instance, consider that an assumption was made at the meta-level. Whenever a contradiction occurs in OLT afterwards, we can declaratively detect it, and then, by means of the *Resume* meta-predicate, we can move back to a previous non contradictory OLT. This can be achieved by a meta-rule, such as

$$\text{If } Assume(\$y) \text{ and } K(\$x,()^{7}) \text{ then } Resume$$

[7]Notice that a contradiction in OLT occurs when OLT contains literals of the form (p,V) and

Assume instances can also be used as conditions to check whether an assumption has been previously made. Meta-level theories keep track of them to allow explicit reasoning about the assumptions active at any moment. We can see the *Assume* action as a pointer (copy of the state) we put in our reasoning process in order to retract what is deduced from it later on, if necessary. An application of the mechanism to a scheduling problem can be found in [Sierra and Godo, 1993].

8 COMMUNICATION CONTROL MECHANISMS

The last issue considered in this Chapter referring to control mechanisms in *Milord II* concerns the communication operation. The object level module process activates the communication operation to either query the user or query some of the submodules. When the operation queries the user, the result is the extension of the current OLT by a propositional variable. However, the communication from a submodule to its present parent module is governed by a set of inference rules concerning the translation between the possibly different corresponding local logics of the modules,[8] and the structural relations concerning the hierarchy. Some of these rules are shown below.

$$\frac{\vdash_{\mathcal{O}_i} (p, V)}{\vdash_{\mathcal{O}} (i/p, \mathcal{T}(V))}$$

$$\frac{\vdash_{\mathcal{M}_i} submodule(j)}{\vdash_{\mathcal{M}} submodule(i, j)}$$

$$\frac{\vdash_{\mathcal{M}_i} submodule(\alpha, j)}{\vdash_{\mathcal{M}} submodule(i/\alpha, j)}$$

$$\frac{\vdash_{\mathcal{M}_i} eval\_type(a)}{\vdash_{\mathcal{M}} eval\_type(i, a)}$$

The first rule translates object level formulas from the submodule $\mathcal{O}_i$ to the module $\mathcal{O}$. The second one informs the module $\mathcal{O}$ that its submodule $\mathcal{O}_i$ has the module $\mathcal{O}_j$ as a submodule. The third one allows us to propagate the KB structure through the module hierarchy. Finally, the fourth rule informs the module $\mathcal{O}$ of the evaluation type of its submodule $\mathcal{O}_i$.

$(\neg p, V')$ such that $V \cap N_n^*(V') = \emptyset$. In this case the literal $(p, ())$ is generated.

[8] The system allows the specification of mappings between local logics in the sense of renaming mappings $\mathcal{T} : A_n \to I(A_m)$ sending each truth-value of an algebra A_n to an interval of truth-values of another algebra A_m, extending in the obvious way to intervals of A_n.

9 EXAMPLES OF COMPLEX REASONING TASKS

9.1 *Scheduling Reasoning System*

This example is an excerpt of the detailed scheduling problem presented in [Sierra and Godo, 1993].

In general, to specify a complex reasoning system in *Milord II* it is necessary to define a hierarchy of modules. This hierarchy captures the usual task/subtask decomposition. However, in some cases it is necessary to iterate over a set of subtasks (for instance, *hypothesis assumption*, *evaluation*, and *revision*). The only way to perform iteration in our language is through the reification/reflection mechanism. This leads to understand the task/subtask decomposition in such cases as a particular relation between the OLTs and the MLTs.

In this scheduling example we build a module in which we associate to each variable a proposition identifier. The space of values for variables is understood in the proposed implementation as the set of truth values of a particular multiple-valued logic. Requirements are expressed as restrictions over the truth value assignments for these propositions. Solutions to the scheduling problem are then considered to be truth value assignments that fulfil the requirements.

In order to implement a scheduling task with a set of requirements to be fulfilled, two modules must be defined:

Requirements module This module will contain the requirements as meta-predicates over the propositions of the object level, i.e. restrictions over the possible values that object level propositions can take. These meta-predicates are defined in the dictionary of the module. This module also defines the particular multiple-valued logic for the object level. In the example there is no truth-values combination, so selecting connectives is irrelevant.

Design task Module This module contains the initial conditions of the problem as object level rules, and the meta-rules that perform the different subtasks of the scheduling process.

Each problem setting is determined by a number of tasks and a set of constrains among them, and requires a particular *requirements module* and a particular *design task module*. In order to be generic all *design task modules* have been programmed with the MLT in common. Thus, to build the actual module that will perform the design, it is necessary to connect this particular generic *design task module* with a concrete *requirements module*, so the former can inherit the requirements of the problem from the later. It is done in the following way, using the refinement operation:[9]

Module Example = Design : Requirements

[9] $A : B$ is a modular expression that generates a new module that results from modifying A by adding elements inherited from B, such as *dictionary* or *logic*. See for details [Puyol-Gruart and Sierra, 1997].

Requirements Module

The requirements of the scheduler under study are:

- The number of tasks to be scheduled.
- The temporal constraint relations among them.
- The number of available time points to perform the tasks.

The tasks are represented as a set of object level propositions $A_1, \ldots, A_n$, the temporal relations as meta-predicates over pairs of elements in the set $\{A_1, \ldots, A_n\}$, and the number of time points $time_1, \ldots, time_q$ as the truth-values of the logic, which will be $\{false, time_1, \ldots, time_q, true\}$.

In our particular case there are four types of constraints that we will represent by four meta-predicates: *before*, *equ*, *diff* and *notbefore*.

1. $before(x, y)$ means that activity x must occur before activity y.
2. $equ(x, y)$ means that activities x and y must occur during the same time period.
3. $diff(x, y)$ means that activities x and y must not occur in the same period.
4. $notbefore(x, y)$ means that activity x must not occur before activity y.

Design Task Module

The implementation of the heuristic search to find a solution to the scheduling problem, is done by defining a set of rules and meta-rules. Rules are responsible for the initial attachment of the whole space of values to the propositions and meta-rules are responsible for the pruning of the search space.

For each proposition A_i representing a scheduling activity a rule like

R00i **if true then conclude** A_i **is** $time_1$

has to be written in order to define the initial possible truth-values for the propositions, i. e. the interval $[time_1, true]$. These intervals represent the root node of the search space. In general the truth-value of the rules determine the initial time point to start the scheduling of the corresponding activity. So initial conditions of the problem can be stated just modifying the certainty values of these rules.

At the meta-level, for each possible constraint violation a meta-rule is written, having as premise a set of conditions that are true when a particular constraint is violated, and as conclusion "how" to restrict the set of possible values for one activity in such a way that the violation is solved. That is, the meta-rule cuts off a set of children states. Meta-rules can be of two types depending on the violation:

1. Meta-rules that restrict the possible values of propositions in such a way that there is no need for backtracking, i.e. only one child remains. This is the case of requirements of type *before*, *equ* and *notbefore*. These meta-rules use the K meta-predicate in their conclusion. An example of such a meta-rule for the *before* requirement is:

   M002 **if** before($x,$y) **and** K($x,**int**($z,true))
   and K($y, **int**($w,true)) **and ge**($z,$w)
   then conclude K($y, **int**(**suc**($z),true))

2. Meta-rules that perform branching, i.e. two children remain. This is the case of constraints of type "diff". These meta-rules have the action *Assume* in their conclusion. Example:

   M005 **if** diff($x,$y) **and** K($x,**int**($z,true))
   and K($y,**int**($z,true)) **then**
   Assume(**list**(($x,**int**(suc($z),true)), ($y,**int**(suc($z),true))))

   A special meta-rule is also needed to detect when no solution is found, and then in that case to backtrack. This situation can be detected when a proposition gets the interval $[true, true]$ as follows:

   M001 **if** K($x,**int**(true,true)) **then Resume**

   When the search space is exhausted and no solution is found, this situation reflects that the set of constraints is inconsistent.

Code of the example

Here we present the complete code of the scheduling module for a set on 4 tasks to be scheduled (see Figure 7). The set of constraints is specific for each example test.[10]

To use the scheduler in a test example we define a module *requirements* which contains the meta-predicates defining the relations between the propositions at the object level, and the set of truth-values representing the admissible time points (see Figure 8).

Now, as said before, using the inheritance property of the operator ":", we define the module that performs a scheduling of four tasks, module *scheduler_test*, with a particular set of requirements, defined in *requirements*.

[10] In the Meta-rules, it is possible to use some system-defined meta-predicates such as: ge (greater or equal), lt (lower than), gt (greater than). The meta-predicates ge, lt, gt can be applied over the order of the truth-values of the local logic, or over the real numbers. It is also possible to perform operations on top of the truth-values, such as suc (successor function), that have to be understood in the context of an ordered set of truth-values.

```
Module Scheduler =
Begin
    Export A1, A2, A3, A4
    Deductive knowledge
        Rules:
            R001 if true then conclude A1 is t1
            R002 if true then conclude A2 is t1
            R003 if true then conclude A3 is t1
            R004 if true then conclude A4 is t1
        Inference system:
            Truth values = (false, t1, t2, t3, true)
    end deductive
    Control knowledge
        Evaluation type: eager
        Deductive control:
        ;; If a propositional variable gets the maximum value
        ;; no solution can be found.
        M001 if K($x,int(true,true))
            then Resume
        ;; X before Y.
        M002 if before($x,$y) and K($x,int($z,true))
            and K($y, int($w,true)) and ge($z,$w)
            then conclude K($y, int(suc($z),true))
        ;; X equal Y
        M003 if equ($x,$y) and K($x,int($z,true))
            and K($y,int($w,true)) and gt($z,$w)
            then conclude K($y, int($z,true))
        ;; X not before Y
        M004 if notbefore($x, $y) and K($x,int($z,true))
            and K($y,int($w,true)) and lt($z, $w)
            then conclude K($x, int($w,true))
        ; X different Y
        M005 if diff($x,$y) and K($x,int($z,true))
            and K($y,int($z,true)) then
            Assume(list(($x,int(suc($z),true)), ($y,int(suc($z),true))))
    end control
end
```

Figure 7. Scheduler module declaration.


```
Module Requirements =
Begin
    Export A1, A2, A3, A4
    Deductive knowledge
        Dictionary:
        Predicates:
            A1 = Name: "A1" Type: many-valued
                Relation: notbefore A4
            A2 = Name: "A2" Type: many-valued
            A3 = Name: "A3" Type: many-valued
            A4 = Name: "A4" Type: many-valued
                Relation: before A2
                Relation: before A3
                Relation: diff A1
        Inference system:
            Truth values = (false, t1, t2, t3, true)
    end deductive
    Control knowledge
        Evaluation type: eager
    end control
end
```

Figure 8. Requirements module declaration.

Module Scheduler_test = scheduler : requirements

Now, let's see the execution trace of the module *Scheduler_test*:

1. Initially OLT deduces the interval [t1,true] for all the propositions A1-A4 by means of rules R001-R004. Upwards reflection operation introduces the following set of predicates into MLT:

$$\mathbf{K}(A_i,\mathbf{int}(t1,true)), \text{ for } i = 1,2,3,4.$$

For the sake of simplicity, in the following we will consider only the minimum value of the interval of truth-values of propositions. This initial situation and all the meta-level processes are represented in Tables 1 to 3.

Table 1. First assumption.

←	**1**	→		←	**2**	→		←	**3**	→
t1	t2	t3		t1	t2	t3	**Assume**	t1	t2	t3
A1				A1				A1		
A2			*M002*	⇒	A2				A2	
A3			*M002*	⇒	A3				A3	
A4				A4			*M005*	⇒	A4	

2. Table 1 represents a part of the meta-level process until the first assumption is generated. Meta-rule M002 is used two times considering the relations "A4 before A2" and "A4 before A3", increasing the value of the propositional variables A2 and A3.

3. Meta-rules M003 and M004 cannot be fired. Given the relation "A4 diff A1" and that A4 and A1 have the same value, an assumption is produced by meta-rule M005:

   Assume(list((A4,**int**(t2,true)), (A1,**int**(t2,true))))

   first the value t2 is assumed for A4.

4. Similarly to the previous meta-level process, Table 2 represents meta-rule actions until a new assumption is performed. M002 increase again the values of A2 and A3.

5. Now a matching occurs for meta-rule M004 because of the relation "A1 notbefore A4", producing a new value for A1.

6. Given that A4 and A1 have the same value, a new assumption is performed by meta-rule M005:

   Assume(list((A4,**int**(t3,true)), (A1,**int**(t3,true))))

   first the value t3 is assumed for A4.

Table 2. Second assumption.

	←	**4**	→		←	**5**	→		←	**6**	→
	t1	t2	t3		t1	t2	t3	**Assume**	t1	t2	t3
	A1			*M004*	⇒	A1				A1	
M002		⇒	A2				A2				A2
M002		⇒	A3				A3				A3
		A4				A4		*M005*		⇒	A4

7. Table 3 represents a new cycle of the meta-level process until a resume operation is performed. Now meta-rule M002 increase again the value of A2, making it equal to *true* (represented as * in the table).

Table 3. Resume.

	←	**7**	→		←	**8**	→
	t1	t2	t3	**Resume**	t1	t2	t3
		A1			*M001*	⇒	A1
M002		⇒	*				A2
			A3				A3
			A4		*M001*	A4	⇐

8. Given that the value for A2 is true, a matching is possible for meta-rule M001 producing a resume operation. Remember the last assumption:

   Assume(list((A4,**int**(t3,true)), (A1,**int**(t3,true))))

   now the value t3 is assumed for A1, and A4 return to the previous value, t2.

   Now no meta-rules can be applied, and the solution is found.

9. Finally downwards reflection operation assigns to the propositions of OLT the solution result:

   {(A1,[t3,true]),(A2,[t3,true]),(A3,[t3,true]),(A4,[t2,true])}

Notice that if the we invert the relation "A4 diff A1" in the code of *requirements module* an equivalent solution is obtained without any assumption.

9.2 *A General Method for solving a class of Default Reasoning problems*

In this section we describe, through an example, a simple approach implemented in *Milord II* to tackle some of the usual problems in defeasible reasoning, such as inheritance, irrelevance and specificity, in a restricted propositional framework.

Consider the following well-known set of defeasible rules:

$$B \to F, P \to B, P \to \neg F$$

The intended behaviour of this set of rules is to infer $\neg F$ given P, to infer F given B, and to infer B given P.

The implementation in *Milord II* makes use of three modules. The module *Penguin* (see Figure 9) defines an object level component with the following characteristics:

```
Module Penguin =
Begin
    Import P, B
    Export F
    Deductive knowledge
        Dictionary:
            Predicates:
                B = Name: "Bird" Type: many-valued
                P = Name: "Penguin" Type: many-valued
                F = Name: "Flies" Type: many-valued
                FPos = Name: "Flies+" Type: many-valued
                    Relation: supports F
                FNeg = Name: "Flies-" Type: many-valued
                    Relation: distracts F
        Rules:
            R001 If B then conclude FPos is d
            R002 If P then conclude B is d
            R003 If P then conclude FNeg is d
        Inference system:
            Truth values = (0, dd, d, 1)
            Conjunction = min
            Modus ponens = Truth table:
                                    ((0  0   0   0)
                                     (0  0   0   dd)
                                     (0  0   dd  d)
                                     (0  dd  d   1))
    End Deductive
End
```

Figure 9. Module Penguin.

- For those propositional variables with contradictory default conclusions a couple of extra propositional variables (in this case FPos and FNeg) that will accumulate the evidence for the particular sign coming from eventually different deductive paths. These propositional variables are related through two relations named *supports* and *distracts*.
- Default rules are written as object level rules with truth-value d, one degree below the maximum one (see next point). Notice that all three rules in the example are considered as default rules, although rules $P \rightarrow B$ and $P \rightarrow \neg F$ could be considered as well as *strict* rules and have attached maximum truth-value 1.

- A *local logic* with as many truth-values as the maximum path in the deductive trees associated to exportable propositional variables. In the current example we take as $\{0 < dd < d < 1\}$ as truth-value set. The combination of conditions (conjunction declaration) is done with the *min* operator. So, shorter paths win. The truth table for the *modus ponens* operation[11] corresponds to the so-called *Łukasiewicz* t-norm and has the characteristic of "counting" the number of applied defaults because it makes the minimum interval value decrease by one term (look at the third column or row in the table).
- At the end of a deductive process, by specialization, the value of the coupled propositional variables is an interval with the minimum value as low as the maximum number of default rules applied to get it.

```
Module Default_interpreter =
Begin
    Control knowledge
        Evaluation type: eager
        Deductive control:
        M001 If K($x, int($min1, $max1)) and supports($x, $y)
                and distracts($z, $y) and K($z, int($min2, $max2))
                and gt($min1, $min2))
                then conclude K($y, int(1,1))
        M002 If K($x, int($min1, $max1)) and supports($x, $y)
                and distracts($z, $y) and K($z, int($min2, $max2))
                and lt($min1, $min2))
                then conclude K(not($y), int(1,1))
        Structural control:
        M001 If K($x, $cert) and supports($x, $y)
                and distracts($z, $y) and K($z, $cert)
                then abort
    End control
End
```

Figure 10. Default Interpreter module.

The *Default interpreter* module (see Figure 10) contains a generic control able to manage any module containing default rules written in the way outlined in the

[11] In this example the algebra of truth-values is different for that defined in Section 3, here defining explicitly the *modus ponens* operator.

module *Penguin*. The connection between the default interpreter module and any containing default rules is done as in the declaration of the module *Solution* (see Figure 11). The *Union module operation*, constructs a new module from two other modules by performing the union component by component. In this example from the modules *Penguin* and *Default_interpreter*. Whenever the union is not feasible an error is raised, for example, when trying to make the union of two modules with different *local logics*. Once connected, the *Default interpreter* module and the *Penguin* module, the new module acts as a module having the deductive knowledge of module *Penguin* and the control knowledge of module *Default_interpreter*. So the execution of *Solution* acts in the following way, because of the eager interpretation defined in *Solution*:

Module Solution = **Union** (Penguin, Default_interpreter)

Figure 11. Solution module.

- P and B are queried to the user. Let suppose P is true.
- An upwards reflection step is performed. Nothing can be deduced.
- $FPos$ and $FNeg$ are deduced. $Fpos$ will have an interval with a minimum value lower than $FNeg$ because two rules were necessary to deduce it. Given $(P, [1, 1])$ and using modus ponens we get $(FPos, [dd, 1])$ and $(FNeg, [d, 1])$.
- An upwards reflexion is performed with $K(FPos, int(dd, 1))$ and $K(FNeg, int(d, 1))$. M002 is applied and $K(not(F), int(1, 1))$ is concluded.
- Downwards reflection produces: $(F, [0, 0])$. As expected penguins don't fly.

9.3 *A Legal Problem: Default Reasoning*

This example is borrowed from Brewka [Brewka, 1994] and it is based on the next statements :

- According to Uniform Commercial Code (UCC) a security interest in goods is perfected by taking possession of the collateral.
- According to Ship Mortgage Act (SMA) security interest in a ship may only be perfected by filing a financing statement.
- UCC is state law, SMA federal law. UUC is more recent than SMA.
- The principle Lex Posterior gives precedence to newer laws.

- The principle Lex Superior gives federal law precedence over state law.
- Miller has possession of a certain ship but did not file a financing statement.

We are interested in formalizing this example in such a way that we can answer negatively the question *Is Millers' security interest perfected?*

In [Brewka, 1994] default logic is enriched by allowing to represent priorities among defaults and reasoning about them. In this formalism, defaults are referenced by a unique name identifier, and preferences among defaults are encoded by a strict partial order, noted $<$, in the set of default names. This preferences are used then to eliminate all those Reiter extensions which are incompatible with the priority information they contain.

Using Brewka's approach, the previous statements are represented as follows.

Defaults:

$UCC : possession \rightarrow perfected$
$SMA : ship \wedge \neg financial\text{–}statement \rightarrow \neg perfected$
$LP(d_i, d_j) : more\text{–}recent(d_i, d_j) \rightarrow d_i < d_j$
$LS(d_i, d_j) : federal\text{–}law(d_i) \wedge state\text{–}law(d_j) \rightarrow d_i < d_j$

Propositional variables and relations:

$possession$
$ship$
$\neg financial\text{–}statement$
$more\text{–}recent(UCC, SMA)$
$federal\text{–}law(SMA)$
$state\text{–}law(UCC)$

The set of Reiter extensions would be:

$E_1 = Th(W \cup perfected, UCC < SMA)$
$E_2 = Th(W \cup \neg perfected, UCC < SMA)$
$E_3 = Th(W \cup perfected, SMA < UCC)$
$E_4 = Th(W \cup \neg perfected, SMA < UCC)$

The only extensions compatible with the priority information defined so far are E_1 and E_2. If we add the next priority information

$$LS(x, y) < LP(y, x)$$

the conflict is solved in favour of E_4.

In Figure 12 an implementation of this example in *Milord II* is presented.[12] Let us comment the more relevant aspects of the code.

[12] The meaning of `set_of_instances(var1, expression, var2)` is the following: given an `expression` containing the variable `var1`, the variable `var2` will be bound to a list containing all the instances of `var1` that make the `expression` true.

```
Module Legal =
Begin
    Import possession, ship, financial_statements
    Export perfected
    Deductive knowledge
        Dictionary:
        Predicates:
        Perfected = Name: "Perfected" Type: many-valued
        Possession = Name: "Possession" Question: "Possession?"
                     Type: boolean
        Ship = Name: "Ship" Question: "Ship?" Type: boolean
        fin_stat = Name: "Financial Statements" Type: boolean
                   Question: "Financial Statements?"
        SMA = Name: "SMA" Type: boolean Relation: law federal
        UCC =  Name: "UCC" Type: boolean
               Relation: law state Relation: more_recent SMA
        Federal = Name: "Federal" Type: class
        State = Name: "State" Type: class
        Rules:
        R001 if possession then conclude UCC is s
        R002 if ship and no(fin_stat) then conclude SMA is s
    End deductive
    Control Knowledge
        Evaluation type: eager
        Deductive control:
        M001 if more_recent($y, $z) then conclude LP($y,$z)
        M002 if law($y, federal) and law($z, state)
             then conclude LS($y, $z)
        M003 if LS($x, $y) and LP($y, $x)
             then conclude prefered(LS($x, $y), LP($y, $x))
        M004 if more_recent($y, $z) and
                set_of_instances($x, prefered($x,LP($y,$z)), $list)
                and equal($list,nil) then conclude prefered($y, $z)
        M005 if law($y,federal) and law($z, state) and
                set_of_instances($x,prefered($x,LS($y,$z)), $list)
                and equal($list,nil) then conclude Prefered($y, $z)
        M006 if K(UCC, int(s,s)) and
                set_of_instances($x,conj(prefered($x, UCC),
                    K($x, int(s, s))), $list) and equal($list, nil)
                and no(K(not(perfected), int(s,s)))
             then conclude K(perfected, int(s,s))
        M007 if K(SMA, int(s,s)) and
                set_of_instances($x,conj(prefered($x, SMA),
                    K($x, int(s, s))), $list) and equal($list, nil)
                and no(K(perfected, int(s,s)))
             then conclude K(not(perfected), int(s,s))
    End control
End
```

Figure 12. Legal module declaration.

- Object level rules are used to model the verification of the conditions of UCC and SMA laws. When they are verified, rules deduce object level predicates named UCC and SMA with the value s (for *sure*), meaning *true* in the many-valued logic used by default in *Milord II*.

- Most important elements are in the meta-level. The first five meta-rules (M001-M005) model, in a straightforward manner, the LP and LS preference criteria.

- The last two meta-rules model the defaults. For example M006 says: If it is known that the conditions of the UCC law are fulfilled, UCC with truth-value *true*, and there are no laws preferred to UCC with their conditions fulfilled, and it is not known the negation of the propositional variable *perfected*, then the propositional variable *perfected* can be assumed.

- The eager evaluation mechanism starts by querying the user about *possession*, *ship* and *financial-statements* in this order, then applies, if possible, the object level rules, upwards reflect, and deduces at the meta-level. Let us follow a trace:

1. Possession? *true*
2. Ship? *true*
3. Financial statements? *false*
4. OL deduction gets: $(ucc, [s, s])$ and $(sma, [s, s])$
5. Upwards reflection produces: $K(ucc, int(s, s))$, $K(sma, int(s, s))$, $more\_recent(ucc, sma)$, $law(sma, federal)$, $law(ucc, state)$, and other irrelevant meta-predicates.
6. M001 applies getting: $LP(ucc, sma)$
7. M002 applies getting: $LS(sma, ucc)$
8. M003 applies getting: $preferred(LS(sma, ucc), LP(ucc, sma))$
9. M004 fails, $\$list$ is not nil
10. M005 applies getting: $preferred(sma, ucc)$
11. M006 fails, $\$list$ is not nil
12. M007 applies getting: $K(not(perfected), int(s, s))$
13. No more meta-rule applies
14. Downwards reflection produces: $(not(perfected), [s, s])$
15. No object level deductions are possible. STOP

10 CONCLUSIONS

It is often the case that reasoning patterns occurring in complex problem solving tasks cannot be modelled (or at least it may turn very cumbersome) by means of a pure logical approach. Extra-logical mechanisms may be of great help in such situations if correctly used in suitable contexts. In this paper we have described the control techniques successfully used in the *Milord II* system. The most remarkable feature is its declarative control which is modelled by a meta-level approach, based on reflection techniques and equipped with a declarative backtracking mechanism. The use of reflection techniques, together with an (implicit) subsumption mechanism at the object level, has been proved specially well suited to tackle the problem of incompleteness of knowledge. As a final remark, it is interesting to notice that, although particular to this system, most of the considered techniques can be of general interest for a variety of multi-language logical architectures (e.g. multi-agent systems).

11 ACKNOWLEDGEMENTS

We acknowledge the Spanish Comisión Interministerial de Ciencia y Tecnología (CICYT) for its continued support through projects: ACRE (CAYCIT 836/86), SPES (880J382), ARREL (TIC92–0579–C02–01) and SMASH (TIC96–1038–C04–01). This research has also been partially supported by the Esprit Basic Research Action number 3085 (DRUMS) and by the European Community project MUM (Copernicus 10053). The definition of this language has profited from ideas coming from many people, among which we would like to thank Francesc Esteva, Ramon López de Màntaras, Jaume Agustí and Don Sannella. Parts of the software have been developed by Josep Lluís Arcos. We also thank the experts that have developed applications based on *Milord II*, Albert Verdaguer, Miquel Belmonte, Marta Domingo, Pilar Barrufet, Lluís Murgui and Ferran Sanz.

This material was previously published in *Future Generation Computer Systems Journal*, pp. 157–172, 1996, and is reproduced with the kind permission of Elsevier.

Lluís Godo, Josep Puyol-Gruart and Carles Sierra
IIIA, Artificial Intelligence Research Institute, Catalonia, Spain.

BIBLIOGRAPHY

[Bacha, 1988] H. Bacha. Metaprolog design and implementation. In *Proceedings of fifth International Conference on Logic Programming*, 1988.

[Brewka, 1994] G. Brewka. Reasoning about preferences in consistency-based nonmonotonic logics. In *Notes of the Workshop on reasoning about incosistency and partiality in dynamic contexts*, 1994.

[Godo and Sierra, 1994] L. Godo and C. Sierra. Knowledge base refinement in *Milord II*. In *Proceedings of 14 th. IMACS World Congress*, 1994.

[Godo *et al.*, 1995] L. Godo, W. van der Hoek, J.J. Ch. Meyer, and C. Sierra. Many-valued Epistemic States. Application to a Reflective Architecture: Milord II. In B. Bouchon-Meunier, R.R. Yager, and L.A. Zadeh, editors, *Advances in Intelligent Computing*, volume 945 of *Lecture Notes in Computer Science*, pages 440–452. Springer-Verlag, 1995.

[Gottwald, 1988] S. Gottwald. *Mehrwertige Logik*. Akademie-Verlag, Berlin, 1988.

[Gottwald, 1993] S. Gottwald. *Fuzzy Sets and Fuzzy Logic*. Vieweg, 1993.

[Hájek, 1995] P. Hájek. Fuzzy logic from the logical point of view. In M. Bartošek, J. Staudek, and J. Wiedermann, editors, *SOFSEM'95: Theory and practice of informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 31–49. Springer-Verlag, Milovy, Czech Republic, 1995.

[Hájek, 1998] P. Hájek. *Metamathematics of fuzzy logic*. Kluwer, 1998.

[Maes, 1988] P. Maes. *Meta-Level Architectures and Reflection*, chapter Issues in computational reflection, pages 21–35. North Holland, 1988.

[Puyol *et al.*, 1992] J. Puyol, L. Godo, and C. Sierra. A specialisation calculus to improve expert system communication. In *Proceedings of the 10th European Conference on Artificial Intelligence, ECAI'92*, pages 144–148, 1992.

[Puyol-Gruart *et al.*, 1998] J. Puyol-Gruart, L. Godo, and C. Sierra. Specialisation calculus and communication. *International Journal of Approximate Reasoning (IJAR)*, 18(1/2):107–130, 1998.

[Puyol-Gruart and Sierra, 1997] J. Puyol-Gruart and C. Sierra. Milord II: a language description. *Mathware and Soft Computing*, 4(3):299–338, 1997.

[Sierra and Godo, 1992] C. Sierra and L. Godo. Modularity, uncertainty and reflection in Milord II. In *Proceedings of 1992 IEEE International Conference on Systems, Man and Cybernetics*, pages 255–260, 1992.

[Sierra and Godo, 1993] C. Sierra and L. Godo. *Formal Specification of Complex Reasoning Systems*, chapter Specifyinf simple schedulling tasks in a reflective and modular architecture, pages 199–232. Ellis Horwood, 1993.