Specifying Simple Scheduling Tasks in a Reflective and Modular Architecture

Carles Sierra, Lluís Godo

Institut d'Investigació en Intel·ligència Artificial (IIIA), CSIC
Camí de Santa Bàrbara, 17300 Blanes, Spain
e-mail: sierra@ceab.es, godo@ceab.es

Abstract

In this paper an architecture, MILORD-II, for complex reasoning tasks is proposed. The architecture allows the definition of a KB as a hierarchical structure of modules. This structure is built by definition of submodules inside modules. Modules are encapsulated, that is, a module has access only to a subpart of its submodules, the one that is specified in their export interface. Each module represents a subpart of the KB and its reasoning system is the result of the interaction through a reification/reflection mechanism of an object and a meta-level reasoning sub-systems. For each module a particular multiple-valued logic can be defined, on top of which the reasoning at the object level is performed. The meta-level logic is a boolean first order logic equipped with two predicates, Ass and Res, that allow both the generation of multiple extensions of object theories and a declarative backtracking mechanism. The formalisation of both object and meta-level languages is presented as well as their interaction and the reasoning dynamics of the whole. Finally, the specification and implementation in MILORD-II of a design reasoning task (scheduling) is described and traces for two test cases are provided.

1 Introduction: A meta-level approach to Complex Reasoning Systems.

Reasoning patterns that appear when modelling complex tasks cannot often be modelled only by means of a pure classical logic approach. This is due to several reasons, incompleteness of the available information, need of using and representing uncertain or imprecise knowledge, combinatorial explosion of classical theorem proving when knowledge bases become large, or a lack of methodology in building complex and large knowledge bases among them. In this paper we present a system (MILORD II) that proposes particular solutions to these problems.
Incompleteness of the available knowledge may lead to make assumptions in order to carry on a deduction process, even if later on these assumptions are proved to be erroneous. (then the reasoning process has to be reconsidered, and previous deductions retracted. This is the kernel of the so-called non-monotonic reasoning patterns). To deal with this problem a meta-level approach, based on reflection techniques and equipped with a declarative backtracking mechanism, is provided by the system. Reflection techniques can also be used to tackle the problem of combinatorial explosion [11].

The use of reflection techniques is a common practice in several knowledge areas, natural language, philosophy, literature, etc. A good general introduction to the topic is [6]. In computer science (CS) and, more particularly in artificial intelligence (AI), reflection has been used in two main areas: Automatic Theorem Proving and Knowledge Based Systems. In fact, the use of reflection techniques is one of the characteristics that differentiate AI development tools from CS environments. Automatic Theorem Proving has found its main limitations in the huge search space it has to deal with. Several approximations have tried to solve this problem: efficient search techniques, limited rule inference sets, limitation of the expressive power of logics, etc. Recently, reflection has been used to implement proof heuristics [9,10]. In this approach the meta-language works with the reified components of the object language (formulas, theorems, axioms, etc.) to build up proof plans that help the prover of the object level language to find the solution.

The application of reflection techniques to KBS has been widely used in the recent past [13,21]. These techniques are used with different representation goals: Implementation of non-monotonic reasoning patterns [18,19], declaration of inference methods [20], architecture description [5]. In general, reflection mechanisms can be understood, in this context, as a clear separation between domain and control knowledge. Few systems have clear semantics, among which we can find OMEGA [13] and BMS [18,19].

Due to the need in AI of dealing many times with uncertain and/or imprecise information, a number of models for representing uncertainty and imprecision have been largely studied within the AI community. In order to have such a possibility of explicitly representing uncertainty, our system provides at the object level a family of representation languages based on multiple-valued logics. These logics will be used in this paper to implement a design reasoning task, but the issue of modelling patterns of uncertain reasoning within the system is out of the scope of this paper. For a deeper insight in this topic, the reader is referred to [2,3].

Modularization is a standard technique to manage the complexity of highly interacting systems, such as KBSs. A module can be understood as a functional abstraction, by fixing both the set of components it needs as input and the type of results it can produce. The internal body of a module can then be hidden from the rest of the system, and the non-desired interactions avoided. This technique has been used in the context of functional programming. Standard ML [12], and Logic Programming [15]. In the architecture proposal of this paper, MILORD II, both techniques, reflection and modularization, are mixed in order to be able to define complex reasoning patterns in the large.

In the next figure you can see how a KB looks like in MILORD II. It consists of a set of hierarchically interconnected modules, each one containing an Object Level Theory (OLT) and a Meta-Level Theory (MLT) interacting through a reflective mechanism. Each module
has also an import/export interface. The user of a KB provides information to modules via the import interface, and modules provide information to the user or to other modules via the export interface.

![Diagram of Milord II KB structure]

Figure 1. Milord II KB structure.

The paper is structured as follows. In the second section the object level, based on multiple-valued logics, is presented and formalised. The next three sections are devoted to the meta-level language, the reification and reflection processes, and the dynamics of the execution of a module respectively. In Section 6 a declarative backtracking mechanism is presented, and in Section 7 a brief description of the MILORD II architecture is made. Finally, the example of the scheduling task proposed in the workshop as an instance of complex reasoning task is presented, followed by the traces of two test cases.

2. The Object level

Reasoning at the object level in one module is generically based on the use of multiple-valued logics. A particular logic can be specified inside a module by defining which is the algebra of truth-values, i.e. which is the ordered set of truth-values and which is the set of operators associated to them. The kind of algebras of truth-values allowed in MILORD-II are those defined as follows:

**Definition**

An ordered Algebra of truth-values is a finite algebra $A_{n,T} = \langle A_n, N_n, T, I_T \rangle$ such that:

1) The ordered set of truth-values $A_n$ is a chain of $n$ elements:

$$0 = a_1 < a_2 < \ldots < a_n = 1$$
where 0 and 1 are the booleans _False_ and _True_ respectively.

2) The negation operator \( N_n \) is an unary operation defined as \( N_n(a) = a_{n+1} \), the only one that fulfils the following properties:

- **N1:** if \( a < b \) then \( N_n(a) > N_n(b) \), \( \forall a \in A_n \), \( \forall b \in A_n \)
- **N2:** \( N_n^2 = Id \).

3) The conjunction operator \( T \) is any binary operation such that the following properties hold \( \forall a, b, c \in A_n \):

- **T1:** \( T(a, b) = T(b, a) \)
- **T2:** \( T(a, T(b, c)) = T(T(a, b), c) \)
- **T3:** \( T(0, a) = 0 \)
- **T4:** \( T(1, a) = a \)
- **T5:** if \( a \leq b \) then \( T(a, c) \leq T(b, c) \) for all \( c \)

4) The implication operator \( I_T \) is defined by residuation with respect to \( T \), i.e. \( I_T(a, b) = \text{Max} \{ c \in A_n \text{ such that } T(a, c) \leq b \} \)

Such an implication operator satisfies the following properties:

- **I1:** \( I_T(a, b) = 1 \) if, and only if, \( a \leq b \)
- **I2:** \( I_T(1, a) = a \)
- **I3:** \( I_T(a, I_T(b, c)) = I_T(b, I_T(a, c)) \)
- **I4:** if \( a \leq b \), then \( I_T(a, c) \geq I_T(b, c) \) and \( I_T(c, a) \leq I_T(c, b) \)
- **I5:** \( I_T(T(a, b), c) = I_T(a, I_T(b, c)) \)

As it is easy to notice from the above definition, any of such truth-value algebras is completely determined as soon as the set of truth-values and the conjunction operator \( T \) are determined. So, varying these two characteristics we can obtain a parametric family of different multiple-valued logics. In this way, each module can have a local type of reasoning, potentially different from others. Additionally, in order to allow a flux of information between modules and submodules with different local logics, renaming mappings between sets of truth values can also be specified inside a Milord II module.

In the same way as in classical logic we distinguish syntactically \( A \) from \( \neg A \) (\( A \) being a formula) because the only two truth value assignments to \( A \) are _true_ or _false_ in our case we need a syntactical representation for each truth value assignment to a formula \( A \). This is necessary to allow explicit reasoning with the whole set of truth-values, i.e., to allow the presence of truth-values in the definition of the inference rules. On the other hand, even in the many-valued setting, partiality in the available information may lead to deal with imprecision at the truth level [7], i.e., one may know that a proposition must take a truth-value but doesn’t know exactly which, and then no single value but a set of possible values (where the truth-value belongs for sure) is given instead. Moreover, having precise information does not always assure (see §2.2) to keep precision through inference processes, namely in Modus Ponens. This kind of imprecision can be handled by attaching intervals of truth values to formulas: the more imprecision there is, the larger the intervals are. In such a calculus the set inclusion relationship accounts for an imprecision ordering in the set of intervals of truth values. Namely, if \( [a, b] \subseteq [c, d] \) we say that \([c, d]\) is at least as imprecise as \([a, b]\), where \( a, b, c, d \) belong to \( A_n \). Then, the language we propose in §2.1 is such that sentences are pairs of type \((p, V)\) where \( p \) is a classical-like sentence and \( V \) is an interval of truth values, possibly empty (represented as [ ]).

180
Moreover, this representation leads to single satisfaction and entailment relations (see §2.2), rather than having classical sentences but several satisfaction and entailment relations, as for example in [18]. Another related approach, based on modal operators instead of truth intervals, is the 3-valued non-monotonic logic NML3 [8].

It is worth noticing that, for instance, partial logics based on Kleene's or Belnap's multiple-valued logics can be modelled inside this framework. Belnap's 4-valued logic corresponds to the Algebra of truth-values with \( A_2 = \{0, 1\} \). Belnap's overdefined value is represented as the empty interval \( \emptyset \), and Belnap's undefined value is represented as the interval \( [0, 1] \).

Another interesting characteristic of the object level reasoning system is that deduction is based on what we call "Specialisation Inference Rule" (SIR), a more general inference rule than Modus Ponens. Such specialisation calculus is introduced in [14] to improve the input/output communication behaviour of the system.

In the following description of the language, the semantics and the deduction system of a particular object level logic, we suppose an algebra of truth values \( A_n, T = \langle A_n, N_n, T, \bot \rangle \) fixed.

2.1. Syntax

For the sake of simplicity, the syntax used in this Object Level Language description is a simplification of the actual MILORD II syntax (see §7). The propositional language \( OL_n = (A_n, \Sigma_O, C, OS_n) \) of the object level is defined by:

- A Signature \( \Sigma_O \), composed of a set of atomic symbols plus true and false.
- A set of Connectives \( C = \{ \neg, \land, \rightarrow \} \)
- A set of Sentences \( OS_n \) whose elements are pairs of classical-like propositional sentences and intervals of truth-values. The classical-like propositional sentences are built from a set of atomic symbols and the above set of connectives, but restricted to literals and rules. That is, the sentences of the language are only of the following types:

\[
\text{Literals: } \text{OLL} = \{ (p, V) \} \\
\text{Rules: } \text{OLR} = \{ (p_1 \land p_2 \land \ldots \land p_n \rightarrow q, V^*) \} \text{ where } p_i \neq p_j, p_i \neq \neg p_j, q \neq p_j \\
\forall i, j \}
\]

where \( p, p_1, p_2, \ldots, p_n \) and \( q \) are literals (atoms or negations of atoms). \( V \) and \( V^* \) are intervals of truth-values. Intervals \( V^* \) for rules are constrained to the upper intervals, i.e. of the form \( [a, 1] \), where \( a > 0 \).

Note: in the sequel we will identify intervals \( [a, a] \) with the value \( a \).

2.2. Semantics

The semantics is basically determined by the connective operators of the truth-value algebra \( A_{n,T} \). Having truth-values in the sentences enables us to define a classical satisfaction relation in spite of the models being multiple-valued assignments.

---

\[1\] The corresponding rule in MILORD II syntax would be:

If \( p_1 \) and \( p_2 \) and \( \ldots \) and \( p_n \) then conclude \( q \) is \( V^* \)
• Models $M_\rho$ are defined by valuations $\rho$, i.e. mappings from the first components of sentences to $A_n$ such that:
  $\rho(\text{true}) = 1$
  $\rho(\text{false}) = 0$
  $\rho(\neg p) = N_n(\rho(p))$
  $\rho(p_1 \land p_2) = T(\rho(p_1), \rho(p_2))$
  $\rho(p \rightarrow q) = I_T(\rho(p), \rho(q))$

• The Satisfaction Relation between models and sentences is defined by:
  $M_\rho \models_0 (p, V)$ iff $\rho(p) \in V$

It is easy to see that the corresponding semantical consequence relationship between sets of sentences and sentences satisfies the following properties:
  SR1: $(p, V) \models_0 (\neg p, W) \iff N^*_n(V) \subseteq W$
  SR2: $(p, V_1) \cup (p, V_2) \models_0 (p, W) \iff V_1 \cap V_2 \subseteq W$
  SR3: $(p_1, \ldots, p_n \rightarrow q, V) \models_0 (p_1 \land \ldots \land p_n \rightarrow q, W) \iff MP^*_T(V_i, V) \subseteq W$,

where $N^*_n$ and $MP^*_T$ are the point-wise extensions of $N_n$ and $MP_T$ respectively. $MP_T$ is a binary function from $A_n$ to the set $I(A_n)$ of intervals of $A_n$ defined as:

$$MP_T(a,b) = \begin{cases} 
  [1], & \text{if } a \text{ and } b \text{ are inconsistent} \\
  [a, 1], & \text{if } b = 1 \\
  T(a, b), & \text{otherwise} 
\end{cases}$$

where $a$ and $b$ are inconsistent if there exists no $c$ such that $I(a, c) = b$.

$MP_T(a, b)$ provides the set of all solutions in $A_n$ for $\rho(q)$ in the following equation system:

$$\begin{cases} 
  \rho(p) = a \\
  \rho(p \rightarrow q) = b 
\end{cases}$$

which corresponds to a multiple-valued version of the classical Modus Ponens inference rule. Property 15 of implication connectives makes useful this function in SR3.

2.3. Deduction system

The object level deduction system is based on the following axiom schemes:

- (AS-1) $\neg \neg p \rightarrow p$, 1
- (AS-2) $p$, $\{0, 1\}$

on the following axioms

- (A-1) $\text{true}$, 1
- (A-2) $\text{false}$, 0

---

1 Actually, $MP^*_T$ is defined to give the minimal interval containing the point-wise extension.

2 In any case any proposition has a truth-value between 0 and 1.
and on the following inference rules:

(RI-1) weakening: \((p, V_1) \vdash_0 (p, V_2)\) where \(V_1 \subseteq V_2\)

(RI-2) not-introduction: \((p, V) \vdash_0 (\neg p, \text{N}_n(V))\)

(RI-3) composition: \((p, V_1), (p, V_2) \vdash_0 (p, V_1 \land V_2)\)

(RI-4) SIR\(^1\): \((p_i, V_i),(p_1 \land p_2 \land \ldots \land p_{i-1} \land p_i \land p_{i+1} \land \ldots \land p_n \rightarrow q, V_i)\)

\(\vdash_0 (p_1 \land p_2 \land \ldots \land p_{i-1} \land p_{i+1} \land \ldots \land p_n \rightarrow q, \text{MP}^*_{\text{T}}(V_i, V_i))\)

From properties SR1, SR2 and SR3 of the semantical entailment, it is easy to check that this deductive system is sound.

**Theorem 1 (Soundness).** Let \(A\) be a sentence and \(\Gamma\) a set of sentences. Then, \(\Gamma \vdash_0 A\) implies \(\Gamma \models_0 A\).

On the other hand, it is also straightforward to check that our deductive system is not complete. For instance, we have \(\{(p \rightarrow q, 1), (q \rightarrow r, 1)\} \vdash_0 (p \rightarrow r, 1)\), but \(\{(p \rightarrow q, 1), (q \rightarrow r, 1)\} \not\models_0 (p \rightarrow r, 1)\). However we can show that the system is complete for literal deduction, i.e., any literal that is semantically entailed by a set of formulas is also deducible from that set of formulas, generalising the literal completeness theorem of [19].

**Theorem 2 (Literal Completeness).** Let \(\Gamma\) be a set of formulas, \((p, V)\) a literal. Then \(\Gamma \models_0 (p, V)\) implies \(\Gamma \vdash_0 (p, V)\).

The proof of these theorems can be found in the long version of [14].

3. The Meta-level

The meta-level language is a classical first order language \(\text{ML} = (\Sigma_{\text{M}}, C, \text{MS})\) defined by:

- A Signature \(\Sigma_{\text{M}} = (\Sigma_{\text{rel}}, \Sigma_{\text{fun}}, \Sigma_{\text{con}}, \Sigma_{\text{var}})\), where

  \(\Sigma_{\text{rel}}\) = A set of atomic predicate identifiers plus \(\text{Ass}, \text{Res}, K, WK\) and \(P\) (with special semantics, see §4+6).

  \(\Sigma_{\text{fun}}\) = A set of classical arithmetic function symbols.

  \(\Sigma_{\text{con}}\) = A set of constants including the truth values and object propositional symbols.

  \(\Sigma_{\text{var}}\) = A set of variable symbols; it can be empty.

- The same set of connectives \(C\) as in the object language.

---

\(^1\) A discussion and motivation of this inference rule can be found in [Puyal et al, 92].
• A set of sentences MS = MLLG ∪ MR where:

1. MLLG is the set of ground literals, in a classical sense, from ΣM.
2. MLR is the set \{p_1 \land p_2 \land ... \land p_n \rightarrow c \mid p_i \text{ and } c \text{ are literals from } ΣM\} of meta-rules, where every variable occurring in \( c \) must occur also in some \( p_i \).

Variables in meta-rules, if any, are considered universally quantified.

The semantics of the language is a first order classical one. The meaning of the special predicates \( K, WK \) and \( P \) will be explained when defining the reification function (§5), which gives sense to these predicates as a representation of object level sentences, and the meaning of \( Ass \) and \( Res \) will be explained when talking about the declarative backtracking mechanism (§6).

The deduction system is based on only one inference rule:

\[ p_1 \land p_2 \land ... \land p_n \rightarrow c, p_1', p_2', ..., p_n', \vdash c' \]

where \( p_1', ..., p_n' \) are ground instances of \( p_1, ..., p_n \) respectively, such that there exists a unifier \( \sigma \) for \( \{p_1 \land p_2 \land ... \land p_n, p_1' \land p_2' \land ... \land p_n'\} \), and \( c' = c\sigma \) is the ground instance of \( c \) resulting from \( \sigma \).

4. Dynamics of the reasoning process

A complex KB consists of a hierarchy of modules. Each module contains an Object Level Theory (OLT) and a Meta-Level Theory (MLT). A module computes values for the propositions contained in its export interface; i.e., it computes the truth intervals for such propositions. A module execution consists of the reasoning process necessary to compute the values for some of the propositions in the export interface, those the user is interested in (the user can actually select a subset of the export interface when he orders the execution of a module).

The execution of a module can 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. So, submodule execution extends the OLTs of modules by adding to them the formulas returned. It is worth noticing that the interaction is made only at the object level (see figure 2).
The reasoning process will be described in terms of variations of the Object Level Theories\(^1\) and Meta-Level Theories. The initial OLT of a module consists of its set of rules, i.e. a partial KB. The same for the MLT, which initially consists of the set of meta-rules.

**Definitions**

- We call an *object-level elementary extension* the extension of the OLT by a single literal. This can be done either by inference on the OLT, or by importing a piece of data either from the user (as specified in the import interface), from a submodule OLT or from the current MLT.
- We call a *meta-level elementary extension* the extension of the current MLT by the set of ground literals resulting from the application of a single meta-rule over all the possible instantiations of its premise.

The communication between OLTs and MLTs is done through reification and reflection processes that are explained in detail in the next section. Here, and as a rough introduction, we can say that a reification process maps sentences of the object level into sentences of the meta-level, whereas a reflection process maps meta-level sentences into object-level sentences, verifying \(\text{reify} (\text{reflect}(A)) = A\).

The reasoning dynamics follows the next schema.

---

\(^1\) Theories are considered not closed under deduction. So, they are closer to the concept of presentation.
STEP 1: The user selects a module to begin the system execution. The current OLT is the set of rules of the module, the current MLT is the set of meta-rules plus the instances of the user defined meta-predicates. These instances are given when defining the dictionary of a module (see §7).

STEP 2: The reasoning process starts at the object level with the current OLT. If no elementary extension of the OLT is produced then STOP, otherwise the reasoning process control is passed to the meta-level (STEP 3).

STEP 3: When the meta-level gets the control it builds the current MLT as the previous MLT plus the axioms coming from the reification of the current OLT. Then the meta-level reasoning process is activated.

If no extension of the MLT can be obtained, the control is passed back to the object-level without extending the current OLT, i.e. the reflection does not modify the current OLT. On the other hand when an inference can be performed, and thus a meta-level elementary extension is made, the control is passed to the object level extending the current OLT by adding the reflection of the computed extension of the MLT¹. In any case the control goes to STEP 2.

This dynamic process goes on till no possible extension of the OLT can be made. Special mention has to be made about the two meta-predicates Ass and Res. These are used to implement hypothetical reasoning patterns as will be described in §6.

5. Reification and reflection

The reification correspondence relates a subtheory of the OLT with the set of ground literals of the meta-language MLL^G.

Definition. We define the minimal literal OLT as

\[ \text{OLT}_{\text{lit}} = \{ (p, W) / p \text{ literal and } W = \bigcap V_i \text{ such that } (p, V_i) \in \text{OLT} \} \]

A small set of meta-predicates are necessary to relate the OLLs with MLL^G. For each one the corresponding reflection rules are defined. Given that the constant names used in the MLT are exactly the same as those used in the OLT as proposition names, the renaming operation, \[ [\ ] \], is omitted.

¹ So, in fact, the object-level will pass immediately the control to the meta-level. The implementation has to take care of the unnecessary passing to and back of the control. This is the implicit iteration construct provided in MLORD II.
5.1 Meta-predicate $K$

$K(p, V)$ means that $V$ is the minimal interval such that the proposition $(p, V)$ belongs to the OLT. There is a close world assumption on this predicate. The reflection rules are:

\[
(p, V) \in \text{OLT}_1 \quad \Rightarrow \quad \Gamma_1 K(p, V)
\]

\[
(p, V) \in \text{OLT}_1 \quad \Rightarrow \quad \Gamma_1 \neg K(p, V)
\]

The reflection process maps the meta level theories into object level literals. Only instances of the $K$ predicate are considered, because the other related metapredicate: $Ass(S)$ is transformed into several instances of the $K$ predicate in different MLT extensions, see §6. The reflection rule that relates MLT with OLT is defined as:

\[
\Gamma_1 K(p, V) \\
\Gamma_0 (p, V)
\]

5.2 Meta-predicate $WK$

$WK(p, V)$ means that $(p, V)$ is deducible in the OLT, i.e. $OLT \vdash_0 (p, V)$. $\neg WK(p, V)$ means that $OLT \vdash_0 (p, V')$ with $V' \neq [0, 1]$ but $V' \not\subseteq V$

\[
(p, V) \in \text{OLT}_2 \quad \text{and} \quad V \subseteq V^* \\
\Rightarrow \quad \Gamma_1 WK(p, V^*)
\]

\[
(p, V) \in \text{OLT}_2 \quad \text{and} \quad V \not\subseteq V^* \\
\Rightarrow \quad \Gamma_1 \neg WK(p, V^*)
\]

5.3 Meta-predicate $P$

$P(p)$ means that $(p, V)$ belongs to the deductive closure of OLT being $V \neq [0, 1]$. If at the moment of the reification the computing of the deductive closure for $p$ is not finished neither $P(p)$ nor $\neg P(p)$ will be generated.

\[
OLT \vdash_0 (p, V) \quad \text{and} \quad V \neq [0, 1] \\
\Rightarrow \quad \Gamma_1 P(p)
\]

\[
OLT \not\vdash_0 (p, V) \quad \text{and} \quad V \neq [0, 1] \\
\Rightarrow \quad \Gamma_1 \neg P(p)
\]
With these reflection rules we can differentiate between different sorts of partiality in the information:
- propositions that have not been still the goal of the prover, \( \{ p \lor \forall V \in \{0, 1\}, \neg K(p, V) \} \),
- propositions provisionally unknown, \( \{ p \land K(p, \{0, 1\}) \} \), and
- propositions that are definitively unknown because they cannot be proven \( \{ p \land \neg P(p) \} \).

6. Declarative Backtracking

There is a main difference between reflecting an instance of predicate \( K \) and an instance of predicate \( Ass \). When we reflect \( K(p, V) \) all future extensions of OLT will contain the fact \((p,V)\), whereas when we reflect \( Ass((p_1, V_1), (p_2, V_2)) \), any future extension of OLT will contain at least one of the elements of the argument set of \( Ass \), i.e. \((p_1, V_1)\) or \((p_2, V_2)\).

More concretely, when a meta-rule with an \( Ass \) predicate in its conclusion is applied, as many extensions of the meta-theory MLT as elements in the argument set of the \( Ass \) predicate 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:

\[ \text{M0001: if } K(p, 1) \text{ and } \neg P(q) \text{ then } Ass(((q, 1), (r, 0))) \]

Suppose also that \( q \) could not be proved in OLT. Then, after the reification process, the current MLT will be the extension of the previous one with the ground literals \( K(q, true) \) and \( \neg P(q) \). So, now the above meta-rule can be applied, and this causes the system to obtain the conclusion \( Ass(((q, 1), (r, 0))) \). The meaning of the meta-predicate \( Ass \) 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 \( Ass \). So in this case we obtain the following two different extensions of the current MLT:

\[
\text{MLT}_1 = \text{MLT} \cup K(q, 1) \\
\text{MLT}_2 = \text{MLT} \cup K(r, 0)
\]
From now on, and until another instance of an Ass predicate or a Res predicate are obtained, the existing communication (reification and reflection) between OLT and MLT is moved to a communication between OLT and MLT1. Thus, in this case, after the reflection process OLT is extended with \((q, 1)\).

In order to backtrack in the tree of MLTs generated by successive applications of Ass predicates, the system provides a special 0-ary predicate Res. When a meta-rule concluding Res is applied, we perform a backtracking in the meta-theories tree. This backtracking restores the father's MLT, and consequently the current OLT becomes the OLT which was active at the moment the assumptions were made by the father's MLT. In the above example, backtracking from MLT1 to MLT makes that \(q\) will not be true in the current OLT, and that immediately the communication (reification and reflection) between OLT and MLT is moved to a communication between OLT and MLT2 (see next figure).

It is worth noticing that predicates Ass and Res provide the system with a declarative backtracking mechanism, similar to the approach taken in MetaProlog [4]. This declarative mechanism allows us to implement several complex reasoning patterns.

For instance, consider that an assumption was made at the meta-level. Wherever a contradiction occurs in OLT afterwards, we can declaratively detect it, and then, by means
of the Res predicate, we can move back to a previous non contradictory OLT. This can be achieved by a meta-rule, such as

\[
\text{If Ass(y) and K(x, (}1\text{) then Res}
\]

Ass can be used as a condition to check if an assumption has been previously made. Meta-level theories keep track of them to allow explicit reasoning about the assumptions active at any moment.

Notice that a contradiction in OLT occurs when OLTs contain literals of the form \((p, V)\) and \(\neg(p, V')\) such that \(V \land N(V') = \emptyset\), and then the RI-2 and RI-3 inference rules are applied to generate \((p, [])\).

It is worth noticing that, in MILORD II, when at the object level deductions may be derived from a reflected fact, \((p, V)\), it is better to use the \(Ass(((p, V)))\) predicate in the conclusion of meta-rules because afterwards and using a \(Res\) meta-predicate we will be able to recover the present state if something goes wrong. Conclusions made by a meta-predicate \(K\) are irrevocable. We can see the \(Ass\) meta-predicate 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.

7. MILORD II Language description

MILORD II is a currently working expert systems shell which contains the features described up to now, and others that are out of the scope of this paper. The interested reader is referred to [1,2,16]. We will concentrate on the description of the modular structure of the language, necessary to understand the next sections of the paper.

The language provides three basic mechanisms of module manipulation:

1) Composition of modules through the declaration of submodules,
2) Refinement of modules, and
3) Composition of modules through operators defined by the user via generic modules definition.

Next, a description of the different components of a MILORD II module is made. Then, generic modules are presented. Afterwards two basic operations on modules are outlined: Combination and Refinement. Combination is used to build up the hierarchical structure of modules. Refinement is used, in what matters here, to perform inheritance between modules. Given the space at hand the MILORD II description will necessarily be just a glimpse.

7.1. Modules

The basic KB units of MILORD II are modules. Modules are compound structures. The basic elements that compose a module are:

---

1 In MILORD II intervals are represented between parentheses instead of brackets.
• Import interface: information to be provided by the user.  
• Export interface: output result of a module.  
• Object level knowledge: an initial Object Level Theory.  
• Meta-level knowledge: an initial Meta-Level Theory.  

Next a detailed description of the basic components of a module and some syntax are presented.

Import interface

Imported facts\(^2\) are those whose values are fixed at run time by the user. The import interface is defined by an expression like:

\[
\text{Import fact}_1, \text{fact}_2, \ldots, \text{fact}_n
\]

No other facts inside the module will receive values from the user. In other words, this set of facts is the only way in which a user can influence the results of a module.

Export interface

Exported facts are those that are visible from outside the module; visible either by users or by other modules. The values of exported facts can be fixed by the OLT, by the MLT or by the user if an exported fact is also an imported one. The export interface is defined by an expression like:

\[
\text{Export fact}_1, \text{fact}_2, \ldots, \text{fact}_m
\]

Facts not mentioned in the export interface are hidden to the user and to the rest of the modules (i.e. their values cannot be consulted). The export interface is the only mandatory component of a module. A module with no exported facts is meaningless.

Object level knowledge

The object level of a module is composed of:

a) Dictionary: this component defines the fact identifiers and some attributes of them, for example their type. It also defines relations between facts, i.e. meta-predicate instances. For example:

---

1 Modules can also import information from other modules via their declaration as submodules. See §7.3.
2 MILORD II object level contains not only multiple-valued propositions, but also numerical variables, variables defined by enumeration and boolean propositions. As a general term we use fact to refer to any of them. A very informal definition of fact might be: an object level representation unit that can receive a value, whatever it is, by the user or by the system.
**Dictionary:**

**Predicates:**

- **Temperature**
  - Name: "Patient's Temperature (in centigrade)"
  - Question: "Which is the patient's Temperature?"
  - Type: Numeric
  - Relation: less-relevant-than AIDS

b) Rules: the set of rules of the object level. For example:

```plaintext
R0005 If fever > 38.5 and shivers then conclude bacterian-disease is sure
```

It is possible to use two special propositions true and false as conditions. These propositions always evaluate to the maximum and minimum certainty values of the multiple-valued logic used in the module respectively. So, it is possible to write rules as:

```plaintext
If true then conclude p is moderately-possible
```

c) Local logic: it is possible to define three main components of an Inference System: (i) the set of truth-values, (ii) a renaming mapping between the truth-values of the submodules, if any, and the truth-values of the module that combines them and (iii) the connective operators used to combine and propagate the truth-values when making inference. A more complete description can be found in [3].

The next piece of code is an example of module definition containing rule and logic declarations.

```plaintext
Module Previous_Treatment =
  Begin
    Import Prev_Treat
    Export Penicillin, Tetracycline
    Deductive knowledge
      Dictionary: not defined here
      Rules:
        R001 If Prev_Treat = (Peni) then conclude Penicillin is sure
        R002 If Prev_Treat = (Peni) then conclude Tetracycline is impossible
      Inference system:
        Truth values = (impossible, sure)
        Connectives^2:
          Conjunction: ((impossible, impossible), (impossible, sure))
```

---

1 Given that rules always have intervals of the type \([a, \infty)\) if these intervals are written in MILORD II as just \(a\).

2 Connectives may be defined, as in this case, as a truth-table, represented by rows, and following the order of the linguistic terms. In the conjunction definition of this example module it is represented that: impossible \& impossible = impossible, impossible \& sure = impossible, sure \& impossible = impossible, sure \& sure = sure, that is, the classical boolean and connective.
Disjunction: (impossible, sure) (sure, sure))

end deductive
end

Meta-level knowledge

The current implementation of the meta-language allows the definition of meta-rules (as explained in previous sections), and the type of module execution, which can be lazy or eager. Lazy means that facts are evaluated, at the object level, only when needed, i.e. imported facts and exported facts of submodules are asked only if they may be useful to compute the export interface of the module. On the contrary an eager module execution obtains, first of all, values for the imported facts and for the exported facts of the submodules and then the deductive knowledge is used. The interaction between OLT and MLT (reification/reflection step) is made after every import information is obtained, after every submodule execution and after every time OLT is extended by deduction.

The next example is taken from the application on scheduling explained in Section 8. In it you can observe the diverse components explained above.

Module Requirements1 =
Begin
  Export A1, A2, A3, A4
Deductive knowledge
  Dictionary:
  predicates:
    A1 = Name: "A1" Type: logic
    A2 = Name: "A2" Type: logic relation: diff A4
    A3 = Name: "A3" Type: logic relation: before A2 relation: before A4
    A4 = Name: "A4" Type: logic
  Inference system:
    Truth values = (false, t1, t2, t3, true)
end deductive
control knowledge
  evaluation type: eager
end control
end

7.2 Generic Modules

Generic modules are the way to define functions from modules to modules. They work in a similar way to a linker in a programming language. An example of a Generic Module is the next one:

Module G(X : S1; Y : S2) : S3 =
Begin
  ::Usual module definition.
  ...
End

This generic module accepts two modules as parameters satisfying the refinement relation with S1 and S2 respectively, and produces a module satisfying the refinement relation with S3 (see §7.4). These generic modules can be defined at top level or at any level in the
hierarchy of modules. In this sense it differs from the treatment of Functors in Standard ML [12]. In the generic module references to parameters X and Y are allowed. The compiler is the responsible of applying the generic modules to their actual parameters, i.e. generic modules are static, they cannot be applied at run time.

7.3. Combination of modules

We say that a module combines other modules when it access to their exported facts. It can be done in two ways: (1) declaring submodules, and (2) applying a generic module. The declaration of submodules is identical, syntactically, to the declaration of modules, and is the key to the hierarchical organisation of KBs. The distinction between a KB and a piece of a KB is lost in MILORD II, and a KB will be then a module, may be containing other modules as submodules.
A generic module can be seen as a module with some submodule names, the name of the arguments, not linked to any particular module. At the moment of the application those submodule names are linked to the particular modules given as parameters (see next figure).

![Figure 5. Generic Module application.](image)

In both cases, i.e. a module which combines its submodules or a generic module applied over some parameters, the resulting module may use, in its rules and meta-rules, the facts appearing in the export interfaces of the modules declared as submodules and/or the parameters. The access to the exported facts is made using a prefix mechanism. An exported fact is identified by two components: 1) a path of module names, separated by "/", indicating how to access to the fact in the hierarchy of modules, and 2) the name of the fact, i.e.

\[ name_1 / name_2 / ... / name_n / fact \_name \]

These combinations of modules build up a hierarchical structure of modules. Each module containing their particular object level and meta-level theories.

7.4. Inheritance through module refinement

Module refinement is a mechanism that allows to incrementally construct knowledge bases. The refinement operation takes two modules and gives another module as result.
The resulting module is essentially equal to the first argument but contains elements of the second argument not defined in the first, i.e. implements a kind of inheritance. Elements that are inherited are dictionary entries and local logics. That is, if the second argument contains a dictionary entry that is not present in the first one, it is inherited. Similarly, if the first argument does not have a local logic definition it is inherited from the second.

The refinement operation is also used to verify if the first argument is an enrichment of the second and to make information hiding [17]. To express that a module C is the result of the refinement between A and B we write

\[ \text{Module } C = A : B \]

or

\[ \text{Module } C : B = A \]

both expressions are equivalent.

7.5 Environment and working procedure

The MILORD II system consists of two main components: (1) a compiler that checks the code of the KB to detect errors, to solve dependencies in the declaration of submodules, to compute the modules resulting from the application of generics, and (2) an interpreter that executes the code generated by the compiler.

![Diagram of MILORD II KB execution procedure.](image)

**Figure 6.** MILORD II KB execution procedure.

The user, when running a KB selects a module and then the interpreter executes it. The interpreter will compute the value for the export predicates of that module and will give them back to the user. In the execution process the user will be asked for the information necessary to compute the results.
8. Specification of the scheduling reasoning system

8.1. Design reasoning systems

In general, to specify a complex reasoning system it is necessary to define a hierarchy of modules. This hierarchy captures the task/subtask decomposition. This is clear in diagnosis reasoning systems (see for example, PNEUMON-1A [22]). However, in a design reasoning system (DRS) the possibility to iterate over a set of subtasks (hypothesis assumption, evaluation, and revision) plays a fundamental role. The only way to perform iteration in our language is through the reification/reflection mechanism. This leads to understand the task/subtask decomposition as a particular relation between the OLTs and the MLTs.

To specify a DRS we build a module in which we associate to each design 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\(^1\). Requirements are expressed as restrictions over the truth values assignments for these propositions. Solutions to the design process are then considered to be truth value assignments that fulfill the requirements.

In order to find a solution, the module representing the design task is executed till a final OLT\(_T^\ast\) is found. This is done through the interaction of the OLT and MLT of the module. The MLT extends the OLT by reducing the set of possible truth values for the propositions (possible models), initially the whole interval \([0, 1]\) of truth values. This reduction is done using heuristics that allow to prune part of the search space. The final OLT\(_T^\ast\) will consist of a set of literals \((p, V)\) where \(p\) is a proposition representing a design variable and \(V\) an interval of truth values. These intervals represent all the non rejected possible values for those variables. When the execution finishes, the truth value assignment \(\rho\) representing a correct solution can be built up from the literals \((p, [v_1, v_2])\) belonging to the final OLT\(_T^\ast\) making \(\rho(p) = v_1\).

In the sequel the modular structure and the different subtask specifications of the DRS in a scheduling framework, as described in the workshop proposal, are described.

8.2. Modular structure of the scheduling reasoning system (SRS)

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

---

\(^1\) This is the approach followed in this paper. Given that MILORD II object language is a bit richer than the object language presented here, we could have done it using numerical variables for example.
defines the particular multiple-valued logic for the object level. However, in the example task 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 (number of tasks and relations between them) requires a particular requirements module and a particular design task module. All design task modules have the MLT in common. Thus, to build the actual module that will perform the design, it is necessary to connect the particular design task module with the 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:

| Module example = design : requirements |

8.3. Subtask specification of the SRS

The subtask decomposition of the scheduling problem is embedded into the design module components. Next the different subtasks and their relation with meta-level constructs will be described.

8.3.1. Requirements Generation

This task is performed by the user through the definition of the requirements module, explained in the previous section, with the particular set of requirements to be fulfilled by the scheduling solutions.

The requirements of the scheduler under study are:
- The number of tasks to be scheduled.
- The temporal relations between them, and
- The number of available time points.

The tasks will be defined 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 \( 0, time_1, \ldots, time_q, 1 \).

In our particular case there are four types of requirements 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 in 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 \( y \).

Meta-predicates are defined (see §7.1) in the dictionary within the definition of the propositions, i.e. the tasks.

So, an schema of the requirements module definition would be:

---

1 Remember that A:B is a modular expression that generates a new module that results from modifying A by adding elements inherited from B, such as the name or logic.
Module Requirements =
Begin
Export A₁, ..., Aₙ
Deductive knowledge
Dictionary:
predicates:

... Aᵢ = Name: "Taskᵢ" Type: logic relation: relᵢ,₁ Aᵢ

... relation: relᵢ,₁⁺m Aᵢ⁺m

... Inference system: Truth values = {false, time₁, ..., time₉, true}
end deductive
control knowledge evaluation type: eager
end control
end

Where relᵢ,₁ stands for an element of {before, equ, diff, notbefore}.

8.3.2. The scheduling search space

The description of the KB that specifies the scheduling task will be done in terms of its associated search space. In this search space the nodes represent the set of possible solutions, that is, all the possible assignments we can build from the proposition truth intervals. Each child represents a subset of the possible solutions of the parent node obtained by reducing the truth-interval of one proposition. There are two ways of doing that: increasing the lower bound or decreasing the upper bound of the truth-interval. The former corresponds to an "as soon as possible" assignment heuristic and the latter to an "as late as possible" assignment heuristic. Depending on the heuristic associated to each meta-predicate we can select for each node the best possible assignment. Once an assignment is chosen for a node, it is evaluated. If no requirement is violated, that assignment is a solution. Otherwise, i.e. if some requirements violations are detected, one of them is heuristically selected to be solved. Then the solution of the selected requirement violation is done by pruning all the children except those that contain no assignment violating that requirement. It is easy to see that all the remaining possible solutions are those contained in the non cut children. If there remains more than one children a search with backtracking has to be performed. Backtracking is done when we are at a leave node and the single solution it represents is found to violate a requirement.

In the considered example this general schema is particularised by:

• The assignment heuristic is "as soon as possible" for all propositions. So, the best assignment for each node is the one with the lowest truth values.

• There are four types of requirements: before, equ, diff and notbefore. The requirement violation of three of them, before, equ, and notbefore, makes that the pruning process leaves only one children, whereas the diff violation leaves two. There are two possible solutions of a diff violation: increase the lower bound of the interval of one of the two conflicting tasks. So, trying to gain efficiency, the implemented heuristics is to solve

---

¹ In our system these heuristics could be specified for each meta-predicate.
any of the former before solving the latter, because in that way branching is generated as late as possible.

8.3.3. Implementation of the heuristic search

The implementation of the heuristic search 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

\[
\text{R0001 if true then conclude } A_i \text{ is time}_i
\]

has to be written in order to define the initial possible truth-values for the propositions, i.e. the interval \([\text{time}_1, \text{true}]\). 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 requirement violation a meta-rule is written, having as premise a set of conditions that are true when a particular requirement violation is produced, 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 requirement violation:

A.- Meta-rules that restrict the possible values of propositions in such a way that there is no need for backtracking, i.e. only one children 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 meta-rules for the before requirement is:

\[
\text{M002 if before}(x,y) \text{ and } K(x, (\text{true})) \text{ and } K(y, (\text{false})) \text{ and } \text{ge}(z, w) \text{ then conclude } K(y, (\text{suc}(z) \text{ true}))
\]

B.- Meta-rules that perform branching, i.e. two children remain. This is the case of requirement diff. These meta-rules have the meta-predicate Ass in their conclusion. Example:

\[
\text{M005 if diff}(x, y) \text{ and } K(x, (\text{true})) \text{ and } K(y, (\text{true})) \text{ then conclude Ass}(((x, (\text{suc}(z) \text{ true})), (y, (\text{suc}(z) \text{ true}))))
\]

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 \([1, 1]\) as follows:

\[
\text{M001 if } K(x, (1 1)) \text{ and type}(x, \text{logic}) \text{ then conclude } \text{Res}
\]

When the search space is exhausted and no solution is found, this situation reflects that the set of requirements is inconsistent.
8.3.4. Relationship between the implementation and the subtask decomposition of the scheduling problem

In the workshop proposal of the Example Reasoning Task a subtask decomposition of the scheduling problem is suggested. It is possible to identify these subtasks in our implementation.

1. Requirements generation. This is a particular module defined by the user, as described in a previous section.

2. Assumption generation. Any time a reflection is made, the truth-intervals assigned to propositions represent all possible assumptions for each of them.

3. Assumption selection. Any time a reification is made, lower bounds of the truth-intervals appearing in instances of predicates $K$ represent the selected assumptions for all the propositions.

4. Design object evaluation. Any time a premise of a meta-rule becomes true, a particular requirement violation is detected. In this case, the solution represented by the current assumptions is not correct. If no premise becomes true, every requirement is fulfilled, and thus the current assumptions are a solution. Given that in this implementation propositions always have an assigned value, the existence of a requirement violation is equivalent to the non-satisfaction of the requirement.

5. Design process evaluation. When a requirement violation is produced, the premise of some meta-rule becomes true, and is activated. Depending on whether there are still solutions to be tested, the possible actions the meta-rule can take are either retract univocally the assumption of a particular proposition (using meta-predicate $K$), or branch over two possible retractions for different assumptions (using meta-predicate $Ass$). Otherwise, the meta-rule takes a backtracking action (using the meta-predicate $Res$).

6. Assumption revision. In the case of a meta-rule deducing an instance of predicate $K$, this subtask is coincident with Assumption Generation. In the case of an $Ass$ predicate, one of the elements of the argument set is selected and considered to be as a $K$ predicate instance, and then we are again in the case of an Assumption Generation. Finally, in the case of a $Res$ predicate, the state in which the most recent $Ass$ predicate has been deduced is recovered and thus, after the reflection is made, the propositions recover the values they had in that state.

8.4. Scheduling Code schema

Next you can see the scheduler schema. This module code collects the different subtasks explained above.
Module Scheduler_n  =
    Begin
    Export A_1, A_2, ..., A_n
    Deductive knowledge
    Rules:
    R001 If true then conclude A_1 is t_1
    R002 If true then conclude A_2 is t_1
    ... 
    R00N If true then conclude A_n is t_1
    end deductive
    Control knowledge
    Evaluation type: eager
    Deductive control:
    M001 If K(x, (1 t)) and type($x, logic) then conclude res
    M002 If before($x,$y) and K($x,($y true)) and K($y, ($w true))
      and ge($z,$w) then conclude K($y, (suc($z) true))
    M003 If equ($x,$y) and K($x,($z true)) and K($y, ($w true))
      and gt($z,$w) then conclude K($y, ($z true))
    M004 If notbefore($x, $y) and K($x,($z true)) and K($y, ($w true))
      and lt($z, $w) then conclude K($x, ($w true))
    M005 If diff($x,$y) and K($x,($z true)) and K($y, ($z true))
      then conclude ass(((x,(suc($z) true),($y,(suc($z) true))))
    end control
    end

9. Examples and tests

9.1. Code of the example

Here we present the complete code of the scheduling module for a set on 4 tasks to be scheduled. The set of requirements is particular for each example test and they will be presented in the next sections.

Module Scheduler_4  =
    Begin
    Export A_1, A_2, A_3, A_4
    Deductive knowledge
    Dictionary
    Rules:
    R001 If true then conclude A_1 is t_1
    R002 If true then conclude A_2 is t_1
    R003 If true then conclude A_3 is t_1
    R004 If true then conclude A_4 is t_1
    end deductive

1 In the Meta-rules, it is possible to use some system-defined meta-predicates such as: ge (greater or equal),
lr (lower than), gt (greater than), type (type of refined fact). The meta-predicates ge, lr, 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.
Control knowledge
Evaluation type: eager
Deductive control:

; ; if a fact gets the maximum value no solution can be found.
M001 if K($x, (1 1)) and type($x, logic) then conclude res
M002 if before($x,$y) and K($x, (true $z true)) and K($y, (true $w true))
and ge($z,$w) then conclude K($y, (true $z true))
M003 if equ($x,$y) and K($x, (true $z true)) and K($y, (true $w true))
; ; X equal Y
and gt($z,$w) then conclude K($y, (true $z true))
M004 if notbefore($x,$y) and K($x, (true $z true)) and K($y, (true $w true))
; ; X not before Y
and lt($z,$w) then conclude K($x, (true $w true))
M005 if diff($x,$y) and K($x, (true $z true)) and K($y, (true $z true))
; ; X different Y
then conclude ass(($x, (true $z true)), ($y, (true $z true)))

end control

end

9.2. First Example

To define the scheduler for the first test example we define a module requirements1 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.

Module requirements1 =
Begin
  Export A_1, A_2, A_3, A_4
Deductive knowledge
  Dictionary:
  predicates:
    A_1 = Name: "A_1" Type: logic
    A_2 = Name: "A_2" Type: logic relation: diff A_4
    A_3 = Name: "A_3" Type: logic relation: before A_2 relation: before A_4
    A_4 = Name: "A_4" Type: logic
  Inference system:
    Truth values = (false, t_1, t_2, t_3, true)
end deductive
control knowledge
  evaluation type: eager
end control

end

Now, and using the inheritance property of the operator ":", we define the module that performs an scheduling of four tasks, module scheduler_test_1, with a particular set of requirements, defined in requirements1.

Module scheduler_test_1 = scheduler_4 : requirements1

Now, let's see the execution trace of the module Scheduler_test_1:

Assumption generation: A_1 (l_1 true) A_2 (l_1 true) A_3 (l_1 true) A_4 (l_1 true)
Assumption selection: A_1 t_1 A_2 t_1 A_3 t_1 A_4 t_1
Design object evaluation: REQUIREMENTS VIOLATION
Design process evaluation: K(A_2, (l_2 true)) K(A_4, (l_2 true))
Assumption revision: A_2 t_2 A_4 t_2
  Increasing the minimum value of the fact A_2 from t_1 to t_2
  Increasing the minimum value of the fact A_4 from t_1 to t_2
Assumption Generation: A_1 (l_1 true) A_2 (l_2 true) A_3 (l_1 true) A_4 (l_2 true)
9.3. Second Example

Similarly to the previous test example we define the set of requirements with a module requirements2.

```
Module Requirements2 =
  Begin
    Export A1, A2, A3, A4
    Deductive knowledge
    Dictionary:
      predicates:
      A1 = Name: "A1" Type: logic relation: equ A3
      A2 = Name: "A2" Type: logic relation: before A3 relation: diff A4
      A3 = Name: "A3" Type: logic relation: equ A1
      A4 = Name: "A4" Type: logic relation: before A3
    Inference system:
    Truth values = (false, t1, t2, t3, true)
  end deductive
  control knowledge
  evaluation type: eager
  end control
end
```

Finally the scheduler for this test example is built.

```
Module Scheduler_test_2 = scheduler_4 : requirements2
```

Now, let's see the execution trace of Scheduler_test_2.

```
Assumption generation: A1 (t1 true) A2 (t1 true) A3 (t1 true) A4 (t1 true)
Assumption selection: A1 t1 A2 t1 A3 t1 A4 t1
Design object evaluation: REQUIREMENTS VIOLATION
Design process evaluation: K(A3, (t2 true))
Assumption revision: t2
Increasing the minimum value of the fact A3 from t1 to t2
Assumption generation: A1 (t1 true) A2 (t1 true) A3 (t2 true) A4 (t1 true)
```
Assumption selection: \( A_1 \) \( t_1 \) \( A_2 \) \( t_1 \) \( A_3 \) \( t_2 \) \( A_4 \) \( t_2 \) 
Design object evaluation: REQUIREMENTS VIOLATION 
Design process evaluation: \( K(A_1, \ (t_2 \ true))) \) 
Assumption revision: \( A_1 \) \( t_2 \) 
Increasing the minimum value of the fact \( A_1 \) from \( t_1 \) to \( t_2 \) 
Assumption Generation: \( A_1 \) \( (t_2 \ true) \) \( A_2 \) \( (t_1 \ true) \) \( A_3 \) \( (t_2 \ true) \) \( A_4 \) \( (t_1 \ true) \) 
Assumption selection: \( A_1 \) \( t_2 \) \( A_2 \) \( t_1 \) \( A_3 \) \( t_2 \) \( A_4 \) \( t_1 \) 
Design object evaluation: REQUIREMENTS VIOLATION 
Design process evaluation: \( Ass(((A_2 \ (t_2 \ true)), \ (A_4 \ (t_2 \ true)))) \) 
Assumption revision: \( A_4 \) \( t_2 \) 
Branching over MLTs: (#:G1089 #:G1090) 
Increasing the minimum value of the fact \( A_4 \) from \( t_1 \) to \( t_2 \) 
Assumption generation: \( A_1 \) \( (t_2 \ true) \) \( A_2 \) \( (t_1 \ true) \) \( A_3 \) \( (t_2 \ true) \) \( A_4 \) \( (t_2 \ true) \) 
Assumption selection: \( A_1 \) \( t_2 \) \( A_2 \) \( t_1 \) \( A_3 \) \( t_2 \) \( A_4 \) \( t_2 \) 
Design object evaluation: REQUIREMENTS VIOLATION 
Design process evaluation: \( K(A_3, \ (t_3 \ true)) \) 
Assumption revision: \( A_3 \) \( t_3 \) 
Increasing the minimum value of the fact \( A_3 \) from \( t_2 \) to \( t_3 \) 
Assumption generation: \( A_1 \) \( (t_2 \ true) \) \( A_2 \) \( (t_1 \ true) \) \( A_3 \) \( (t_3 \ true) \) \( A_4 \) \( (t_2 \ true) \) 
Assumption selection: \( A_1 \) \( t_2 \) \( A_2 \) \( t_1 \) \( A_3 \) \( t_3 \) \( A_4 \) \( t_2 \) 
Design object evaluation: REQUIREMENTS VIOLATION 
Design process evaluation: \( K(A_1, \ (t_3 \ true)) \) 
Assumption revision: \( A_1 \) \( t_3 \) 
Increasing the minimum value of the fact \( A_1 \) from \( t_2 \) to \( t_3 \) 
Assumption Generation: \( A_1 \) \( (t_3 \ true) \) \( A_2 \) \( (t_1 \ true) \) \( A_3 \) \( (t_3 \ true) \) \( A_4 \) \( (t_2 \ true) \) 
Assumption selection: \( A_1 \) \( t_3 \) \( A_2 \) \( t_1 \) \( A_3 \) \( t_3 \) \( A_4 \) \( t_2 \) 
Design object evaluation: EVERY REQUIREMENT IS FULFILLED 
Design process evaluation: NIL 
Assumption revision: NIL 

---FINAL ANSWER----- 
THE FACT \( A_1 \) IS \( t_3 \) 
THE FACT \( A_2 \) IS \( t_1 \) 
THE FACT \( A_3 \) IS \( t_3 \) 
THE FACT \( A_4 \) IS \( t_2 \) 

9.4 Summary of the executions

In the next figure you can see the results obtained by the execution of the two sets of requirements proposed in this workshop.

Figure 7. Correct scheduling for the first example (left) and the second one (right).
As a glimpse of the complexity of the proposed solution we can say that the complexity of the scheduling task is \( O(n^2 \cdot m \cdot p^2 \cdot 2^q) \) where \( n \) is the number of before relations, \( m \) is the number of equ relations \( p \) is the number of notbefore relations and \( q \) is the number of diff relations. The complexity is considered as the number of proposition value modifications necessary to get a solution.

In the two examples the number of reification/reflection steps is 2 and 5 for the first and the second examples, respectively. The heuristic of applying first the meta-rules that solve the before, equ and notbefore relations, seems to give an acceptable concrete complexity of the scheduler proposed. Further study seems, however, necessary.

10. Conclusions

In this paper, MILORD II, a meta-level architecture for complex reasoning tasks, has been presented. Its main characteristics are the multiple-valued local logics management, the reification/reflection processes and the hierarchical modular structure. The object and meta-level languages are formalised and their dynamic interaction described. A glimpse of the whole MILORD II shell is also given. MILORD II\(^1\) has been used to implement the scheduling reasoning pattern proposed in the call for the ECAI'92 Workshop on Formal Specification Methods for Complex Reasoning Systems.

11. Bibliography


\(^1\)MILORD II is implemented on top of Macintosh machines and it is programmed using CommonLisp